Joe Pritchard

Descubre tu MSX

Programación y Aplicaciones

AM
VIFI
ANAYA

MSX

DESCUBRE TU MSX

Programación y aplicaciones

Descubre tu MSX

Programación y aplicaciones

Joe Pritchard



ANAYA MULTIMEDIA

MICROINFORMATICA

Título de la obra original:
MSX EXPOSED

Traducción de: Laura Feyto y Javier Luis de los Mozos
Diseño de colección: Antonio Lax
Diseño de cubierta: Narcís Fernández

Edición publicada por acuerdo con Melbourne House (Publishers) Ltd., Londres.

Villafranca, 22. 28028 Madrid
Depósito legal: M. 29.623-1985
ISBN: 84-7614-035-5
Printed in Spain
Imprime: Anzos, S. A. - Fuenlabrada (Madrid)

Indice

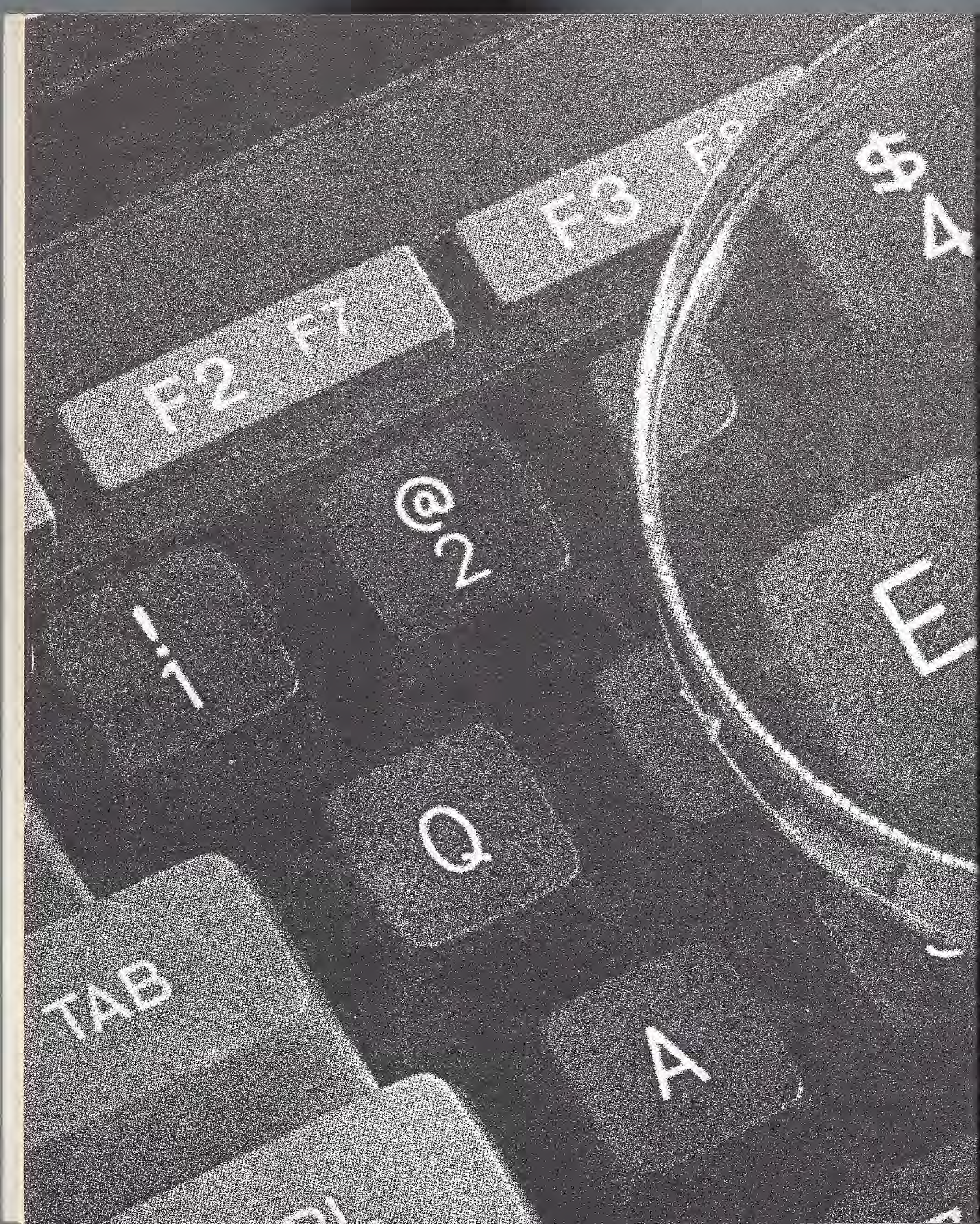
Prefacio

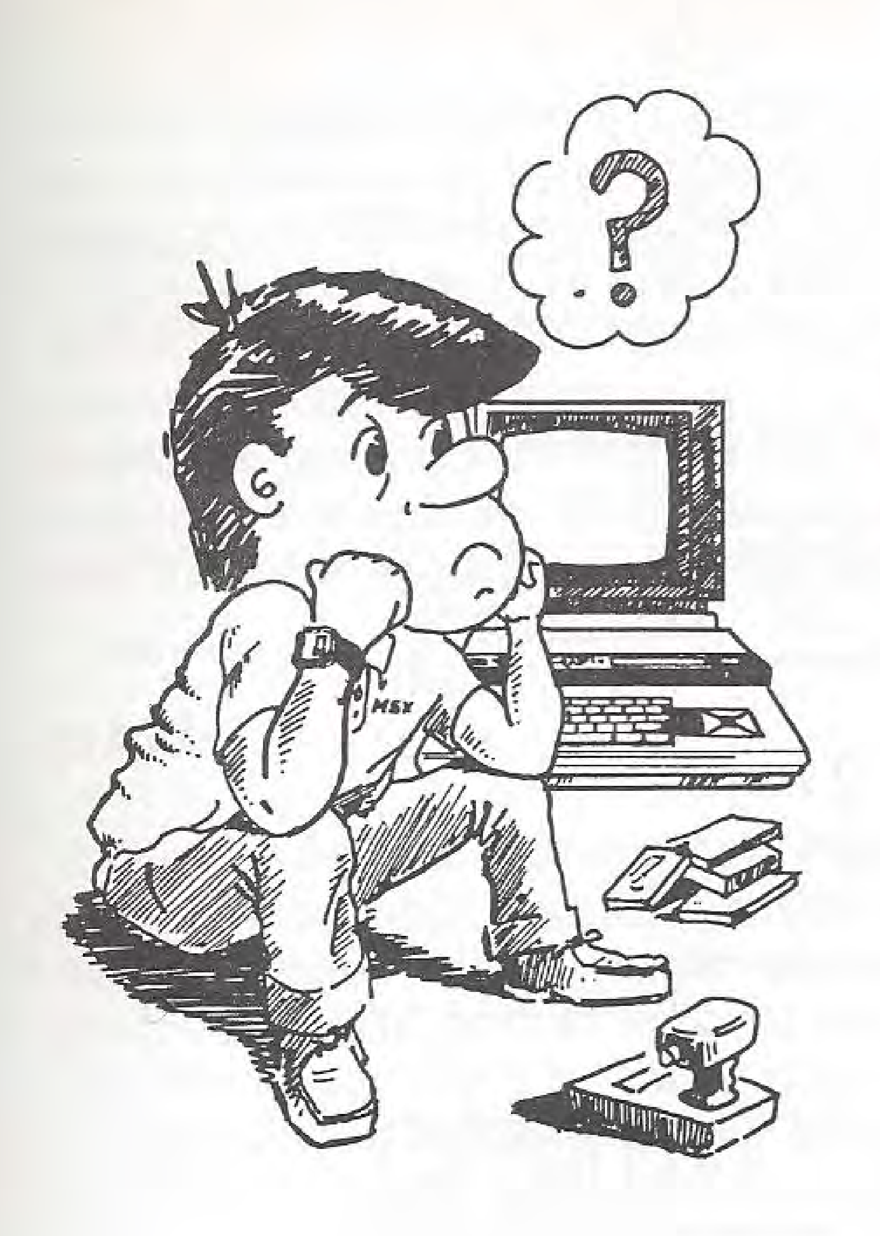
En este libro se da al programador de MSX una visión de los distintos instrumentos que permiten que el ordenador MSX pueda ser programado para obtener el máximo beneficio de la máquina. Muchas de las rutinas de demostración para acceder directamente a los distintos componentes del sistema están escritas en BASIC, pero las principales pueden ser traducidas directamente a código máquina.

Lo que no se ha intentado hacer es enseñar al programador el código máquina del Z80, ya que este tema ocuparía un libro entero por sí solo. En su lugar se da una visión particular de cómo el programador del código máquina puede obtener acceso a los componentes del sistema MSX tan fácilmente como si el programa estuviera escrito en BASIC.

Me gustaría expresar mi gratitud a todos aquellos que han estado involucrados, directa o indirectamente, en la producción de este libro: a Alfred Milgrom y su maravilloso equipo; al doctor Ian Logan, por sugerirme la estructura del libro; a mi familia —especialmente a mis padres y a mi tío—, que ha colaborado con su discreción y escasez de correo y visitas durante la preparación del libro, y a mi esposa, Nicky, que ha colaborado soportando mi ausencia durante varias horas por un largo período de tiempo para escribir y mecanografiar el libro. Esta obra está dedicada a todas estas personas, así como al personal de South Yorkshire & Humberside Microelectronics Education Programme Regional Centre. Y, por último, a doña H. M. A. Brownlow, por razones obvias para aquellos que me conocen.

JOE PRITCHARD
Doncaster, 1984


F2 F7
F3
$
4
@
2
!
1
E
Q
TAB
A



1 El sistema MSX[1]

El sistema MSX es un concepto totalmente nuevo para los ordenadores personales; ¡una serie de ordenadores que son compatibles unos con otros en términos de lenguaje BASIC y características técnicas! Cualquier ordenador que proclame el estándar MSX será capaz de ejecutar el *software* que ha sido escrito para otras máquinas del mismo sistema, y tendrá acceso asimismo a un extremadamente amplio abanico de *software*. El estándar MSX fija un sistema mínimo al que deben ajustarse todas las máquinas, minimizando o desterrando totalmente los temores de que el *software* escrito para que una máquina funcione en otras. Una vez que la máquina posee esta configuración mínima, los fabricantes añadirán individualmente características específicas, que incluirán elementos como la electrónica necesaria para manejar una impresora y *joysticks*. No obstante, ciertos elementos del sistema siempre serán constantes, y son éstos los que en este libro se describen y exploran. Comencemos examinando el sistema como un todo.

Implementación MSX mínima

La figura 1.1 muestra un diagrama de bloques del "interior" de una máquina MSX. No daremos aquí una descripción profunda de cada uno de los principales componentes del sistema, sino en capítulos posteriores. Sin embargo, veamos superficialmente cada componente del sistema para apreciar cómo se ajusta al resto.

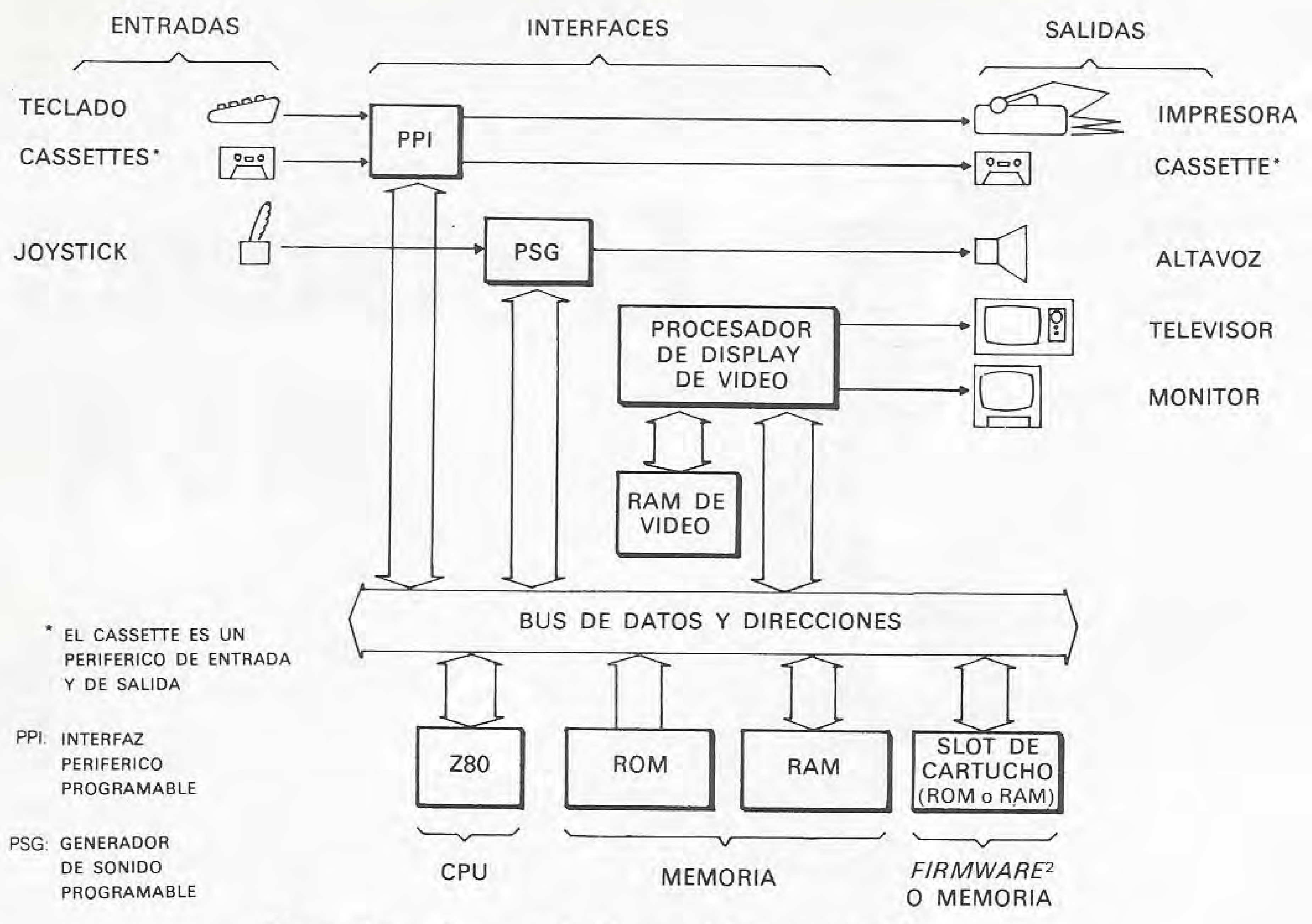


Figura 1.1.—*Esquema de bloques del sistema MSX.*

CPU Z-80

Es el corazón del sistema MSX, la unidad central de proceso (*Central Processor Unit*). Es un *chip* microprocesador, un ingenio electrónico que, junto con otros componentes, controla las acciones del resto del sistema. Puede ser gobernado por una serie de instrucciones que se realizan según una secuencia almacenada. A esta secuencia de instrucciones se le llama PROGRAMA. Trataremos más de cerca este elemento en capítulos siguientes; por ahora es suficiente decir que cualquier ordenador de la serie MSX tendrá este *chip* como CPU. La CPU está continuamente ejecutando el programa contenido en la ROM MSX, y es este programa de ROM, el intérprete BASIC, el que ejecuta su programa BASIC. Usted puede ordenar a la CPU ejecutar otros programas que han sido escritos en un lenguaje llamado CODIGO MAQUINA[3], usando un comando BASIC llamado USR. Trataremos este comando con más detalle en los capítulos concernientes a la inclusión del código máquina en sus programas BASIC.

ROM

Significa "memoria solamente de lectura", es abreviatura de *Read Only Memory*. Contiene un programa que es ejecutado por la CPU siempre que se pone la máquina en funcionamiento. El programa almacenado en esta área de memoria es permanente

e inalterable por el usuario, y contiene las instrucciones necesarias para que la CPU sea capaz de leer el teclado, ejecutar sus programas BASIC y realizar docenas de otras tareas que su ordenador MSX debe hacer. Cualquier ordenador de la serie MSX tendrá una ROM que es muy similar a la de su máquina, si no idéntica. Es una ROM de 32K[4] que posee espacio para almacenar más de 32.000 números diferentes, teniendo cada uno un valor entre 0 y 255 y representando una instrucción para el Z-80, o parte de una instrucción o de un dato que la CPU puede necesitar para realizar sus tareas. En el apéndice se dan más detalles acerca de "K", y números que, asimismo, explican los sistemas de numeración.

RAM

Es otra área de memoria del ordenador, pero de un tipo diferente de la ROM. Se la conoce como "memoria de acceso aleatorio" *(Random Access Memory)*, y el usuario puede modificar su contenido sin dificultad. Aquí es donde se almacena su programa BASIC cuando lo ha tecleado, y es también la memoria que utiliza el ordenador como "papel en sucio" cuando está ejecutando sus programas. Cualquier variable que declare su programa BASIC mientras se esté ejecutando es almacenada en RAM. Ordenes como CLEAR y NEW afectan a la RAM; CLEAR inicializa todas las variables numéricas a cero y todas las cadenas *(strings)*[5] vacías, afectando directamente al área de RAM que contiene las variables. NEW borra un programa de la memoria del ordenador, también afectando directamente el contenido de la RAM. El comando BASIC POKE permite al usuario modificar la RAM, como veremos cuando lleguemos a escribir programas en código máquina. Por último, hay una manera bastante drástica de modificar la RAM: ¡desconectar el ordenador!

La mínima cantidad de RAM que un sistema MSX puede tener es 8K, y usted puede añadir memoria a su sistema utilizando los *slots*[6] de ampliación. En el capítulo de mapas de memoria se darán más detalles acerca de los *slots* de ampliación.

"Slot" de ampliación

Es una característica vital en el concepto MSX, y todos los ordenadores MSX serán totalmente compatibles en ese aspecto. Mediante el *slot* puede añadirse de varias maneras memoria al ordenador. Hay cuatro *slots* en el sistema MSX mínimo, y es posible añadir más. Por el momento, sólo describiremos los *slots* como un medio de añadir más ROM o RAM al sistema.

RAM de video

Es un tipo especial de RAM que está dedicada totalmente a mantener la información usada para el *display* de video. Mientras que la RAM normal es directamente accesible por la CPU, la RAM de video no. La CPU altera las posiciones de memoria en la RAM de video usando el procesador de *display* de video. Hay 16K

de RAM de video, comúnmente abreviado VRAM (esta abreviatura será usada en todo el libro).

Procesador de "display" de video

Es otro circuito integrado que cualquier ordenador del estándar MSX debe poseer. El procesador de *display* de video, o VDP *(Video Display Processor)*, es un *chip* dedicado al control del *display* en video o televisión del ordenador. En la serie MSX, el circuito integrado usado es el TMS9918A o un *chip* muy similar. El VDP es dirigido por la CPU, y proporciona cuatro tipos diferentes de pantalla. Estos tipos de pantalla se llaman MODOS de *display*, y serán estudiados con más detalle en un capítulo posterior. El VDP interconexiona el ordenador con la unidad de *display* que ha de ser usada con este último. Toda la información que se requiere para crear la pantalla está contenida en VRAM, y la manera en que el VDP interpreta estos datos depende del modo de *display* en uso. La VRAM es modificada siempre que el usuario escribe algo en la pantalla, utiliza los comandos CLS, COLOR, VPOKE o alguno de los relacionados con *sprites*.

Generador de sonidos programable

Dentro de los circuitos integrados primordiales en el concepto del sistema MSX, éste es el tercero en la lista, y en todas las máquinas debe ser compatible con el AY-3-8910. La abreviatura que emplearemos para el generador de sonido programable es PSG *(Programmable Sound Generator)*, y es responsable del amplio rango de sonidos que pueden ser obtenidos de los ordenadores MSX. El PSG interactúa directamente con la CPU y es controlado por ella. También es responsable de la implementación de las funciones de entrada/salida[7] *(input/output)* del sistema a las que tiene acceso el usuario, generalmente en la forma de *joysticks*.

Sistema de entrada/salida ("input/output")

La CPU necesita ser capaz de interactuar con otros periféricos aparte del *display* de video y del generador de sonidos. Entre ellos se incluyen el cassette, el teclado, e impresoras. El interfaz[8] entre estos periféricos y la CPU es el *chip* programable de entrada/salida a los periféricos, cuya abreviatura es PPI *(Programmable Peripheral Input/Output)*. El PPI debe ser compatible con el circuito integrado 8255. Y es controlado por la CPU.

Estos son los componentes esenciales de los ordenadores MSX. En capítulos siguientes de este libro describiremos cómo cada componente se ajusta dentro del sistema, dándole al usuario una visión de la función de cada uno de ellos en el ordenador. Sin embargo, veamos brevemente cómo están conectados entre ellos estos componentes. La figura 1.1 muestra la manera en que los componentes del sistema interactúan entre ellos; pero... ¿cómo pasan exactamente los datos entre, digamos, la CPU y la ROM?

El sistema de BUS

En informática, el BUS no es el enorme vehículo con ruedas que todos nosotros solemos echar de menos en las paradas de autobús. Un *bus* en un ordenador es un conjunto de conductores eléctricos que transportan señales eléctricas representando dígitos binarios 0 ó 1; las señales se manejan en el ordenador en forma binaria, y estos ceros o unos se llaman BITS. Si está interesado en el sistema binario, vea el apéndice al final del libro. Ocho de estos bits conforman un byte, que constituye la unidad básica de datos transferida por la CPU. Estos bytes pueden representar números que toman valores entre 0 y 255, y todos los números que usa la CPU se representan por bytes.

El *bus* de datos, que es el medio de transferir información de la CPU a la memoria, al PSG o al VDP, lleva los bytes por todo el ordenador (cada byte representa un número), siendo colocados secuencialmente en el *bus* de datos.

¿Cómo sabe el ordenador qué hacer con un número del *bus* de datos? Bien; hay un segundo *bus* en el ordenador llamado *bus* de DIRECCIONES *(address bus)*, que es de 16 bits. Por ello puede llevar números entre 0 y 65535, y cada uno de estos números se refiere a una cierta posición de memoria del ordenador. Cuando coloca un byte en el *bus* de datos la CPU también coloca, en el *bus* de direcciones, la de la posición de memoria en que ha de ser escrito. Esta dirección se puede referir a una posición de memoria o a una parte del PPI, del VDP o del PSG del sistema.

Un tercer *bus*, llamado *bus* de control, controla todas estas transferencias de datos e informa de los periféricos conectados a los *bus* de datos y de direcciones si la CPU quiere leer o escribir datos en alguno de ellos.

El último *bus* del ordenador es el que conecta el VDP a la VRAM. Este sistema de *bus*, que consiste en uno de datos y uno de direcciones, no es utilizable por la CPU: sólo es accesible por el VDP.

Hemos visto hasta aquí los principales componentes del sistema MSX. En el siguiente capítulo examinaremos los aspectos más simples del BASIC MSX. Si ha tenido dificultad con alguno de los conceptos introducidos, no se preocupe. Ideas tales como expresiones, variables y constantes se explicarán en el capítulo 3.

NOTAS DEL CAPITULO

1.—MSX: Abreviatura de *Microsoft Super eXpanded.*

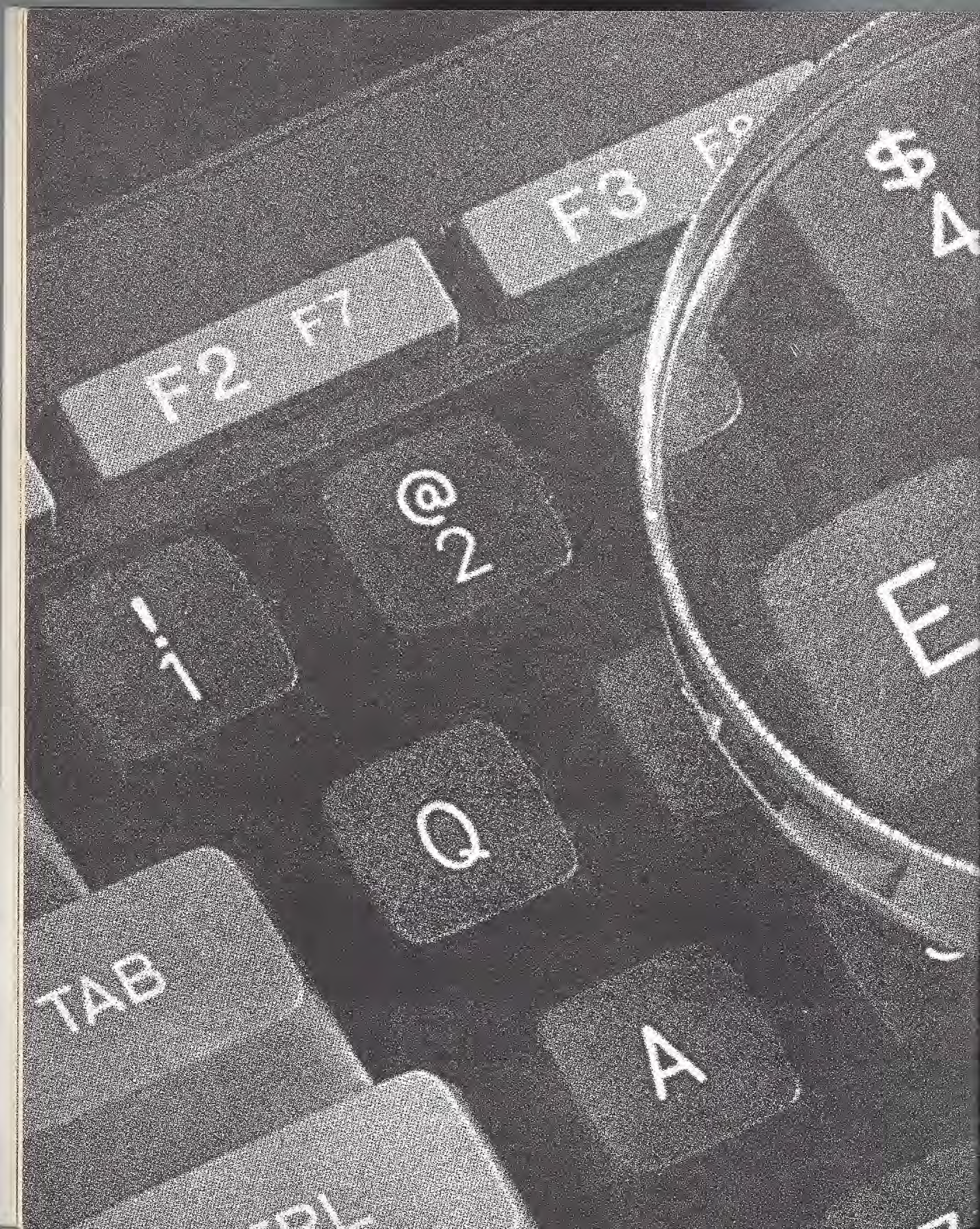
2.—FIRMWARE: Microprogramas colocados en memoria ROM para resolver un problema particular. Se les considera como un soporte de programación inalterable. Es un conjunto de instrucciones no cambiables. Puede considerarse como un punto intermedio entre *software* y *hardware*.

3.—CODIGO MAQUINA: Es el único lenguaje que entiende la CPU. Está compuesto de instrucciones representadas por números (ceros y unos). Es el lenguaje de más bajo nivel, lo que quiere decir que no es similar al lenguaje humano. El intérprete de la ROM MSX se encarga de traducir las instrucciones BASIC a código máquina.

4.—K: 1K equivale a 1.024 bytes, y 1 byte equivale a 8 bits.

5.—STRING: Secuencia de caracteres (alfanuméricos o especiales) que son tratados como un dato único. La longitud del *string* es el número de caracteres que contiene.

6.—SLOT: Ranuras donde se insertan los cartuchos de ampliación.
7.—ENTRADA/SALIDA: Traducción de *input/output*. En el libro se utiliza la expresión inglesa por ser la más corriente en el lenguaje informático.
8.—INTERFAZ: Elemento del sistema que interconexiona la CPU o el ordenador con otros componentes auxiliares del sistema; a estos componentes se les llama "periféricos".


F2 F7
F3
$
4
@
2
!
1
E
Q
TAB
A



2
El BASIC esencial

En este capítulo encontrará descripciones escuetas de todas las instrucciones BASIC comunes. Tales instrucciones no son específicas del BASIC MSX; por eso, a los comandos y sentencias listados en este capítulo se les ha llamado "instrucciones esenciales". Ninguno de los comandos listados en este capítulo está relacionado con las excelentes capacidades gráficas o de sonido de los ordenadores MSX; estas características serán descritas con detalle en capítulos posteriores. La función de éste es proveer una guía de referencia rápida de las principales sentencias BASIC que se usan en todos los programas.

Los primeros comandos que veremos son aquellos que se usan en modo directo, o sea, sin número de línea. Son instrucciones que están dedicadas principalmente a ayudarnos a escribir programas, así como a listarlos.

AUTO n, m

Este comando genera los números de línea automáticamente. Puede ser llamado tecleando la palabra, o usando la tecla de función F2. El primer número de línea generado es *n*, y los siguientes están separados por el incremento *m*. Así:

```
AUTO 10,10
```

generará números de línea 10, 20, 30, etc. El número de línea más alto admisible en las máquinas MSX es 65529, y si AUTO genera uno más alto, o si usted lo teclea,

el mensaje de error *Syntax Error* (error de sintaxis) será generado. Si el número de línea generado por AUTO está ya ocupado por sentencias BASIC, entonces un "*" es visualizado después del número de línea. Tecleando RETURN[1] en este punto conservará cuanto hubiera en esa línea. La función de AUTO puede ser anulada pulsando CTRL-C o CTRL-STOP. AUTO *n* generará números de línea, comenzando en la línea *n*, con el último incremento especificado.

Con frecuencia resulta útil dejar líneas en blanco para separar del programa una parte de otra. Si teclea simplemente un número de línea seguido por un espacio en el sistema MSX, la línea no es insertada. No obstante, se puede poner al principio de una línea ";", que aparece y no causa problemas. Así:

```
100 :
```

dejará una línea en blanco exceptuando ":" en la línea 100. Si teclea en una línea sin espacios, por ejemplo

```
10REM
```

se insertarán los espacios necesarios en los lugares apropiados de la línea, resultando 10 REM.

CONT

Este comando (su nombre es abreviatura de *Continue*) prosigue la ejecución, si el programa se ha parado, usando CTRL-STOP. Para que continúe el programa, teclee el comando y pulse RETURN o use la tecla de función F8. Si el programa ha terminado normalmente su ejecución, CONT no tendrá efecto, pues no puede ser usado como sentencia en una línea de programa. También, si la interrupción ha sido causada por una condición de error, CONT no será efectivo. Finalmente, no trabajará si el programa ha sido editado una vez que se pulsó CTRL-STOP.

DELETE n-m

Este comando nos permite borrar bloques de líneas del programa. *n* es el primer número de línea del bloque a borrar y *m* el último. DELETE *n* sólo borrará del programa la línea *n*. DELETE -*m* borrará todas las líneas de programa entre 0 y la línea *m*, ambas inclusive. Así:

```
DELETE -100
```

borrará todas las líneas del programa comprendidas entre la 0 y la 100, ambas inclusive. DELETE 100-, que se podría esperar borrase todas las líneas del programa entre la línea 100 y el final, no está permitido. Si el parámetro *m* es menor que el *n*, como en

```
DELETE 300-200
```

se generará un error. También ocurrirá esto si las líneas referenciadas por la instrucción no existen. DELETE puede ser usado como parte de una línea de programa, pero tan pronto como la operación de borrado haya sido completada, el ordenador interrumpirá la ejecución del programa y retornará a modo directo.

LIST n-m[2]

Le permite ver el programa que está escribiendo. *n* y *m* son números de línea, siendo *n* el más bajo que nos interesa y *m* el más alto. Si se usa el comando LIST sin ningún parámetro, la totalidad del programa es listada en la pantalla. Parece obvio pensar que el listado desaparecerá por su parte superior antes de que haya terminado de leerlo; sin embargo, una simple presión de la tecla STOP causará la interrupción del proceso. El proceso proseguirá cuando presione la tecla STOP por segunda vez. Esta operación puede repetirla con la frecuencia que se requiera. Si se usan parámetros, el comando trabaja como se explica a continuación:

LIST *n*	Lista la línea *n*.
LIST *n*-	Lista de la línea *n* al final del programa.
LIST -*n*	Lista desde el principio del programa hasta la línea *n*.
LIST *n*-*m*	Lista las líneas *n* a la *m* del programa, ambas inclusive.

El listado puede ser interrumpido por CTRL-STOP. LIST puede ser una sentencia en un programa; pero, tan pronto como el listado haya terminado, el ordenador entra en modo directo y cesa la ejecución del programa.

LLIST

Este comando es similar a LIST, pero envía el listado a la impresora si ésta está conectada. Si no lo está y se emite el comando, el ordenador quedará bloqueado hasta que se pulse CTRL-STOP. Entonces le retornará el control con el mensaje *Device I/O Error*. Los parámetros que se pueden dar a este comando son los mismos que los de la instrucción LIST.

NEW

Este comando borra el programa BASIC residente en la memoria del ordenador. Puede ser incorporado en una línea de programa, pero, como borra éste de la memoria tan pronto como sea ejecutado, no existen muchas aplicaciones para él en este modo.

RENUM

Este comando renumera los programas, manteniendo la secuencia en que las líneas aparecen en el programa, pero alterando los números de línea actuales que las sentencias poseen. La sintaxis del comando es

```
RENUM nuevo,antiguo,incremento
```

nuevo

Es el primer número de línea a usar en la nueva secuencia.

antiguo

Es el número de línea existente actualmente, que se va a usar como lugar de comienzo de la operación de renumeración.

incremento

Es el "hueco" que se debe dejar entre las líneas de programa adyacentes.

Los parámetros antiguo e incremento no son obligatorios; si se omiten, el ordenador asume un valor de 10 para ambos. A continuación se expone un ejemplo de su uso:

```
1 REM 1
2 REM 3
3 REM 4
```

Para renumerar ahora este ejemplo, usando RENUM 10, obtendríamos:

```
10 REM 1
20 REM 3
30 REM 4
```

Que, renumerado con el comando RENUM 10,10,100, nos dará:

```
10 REM 1
110 REM 3
210 REM 4
```

Como con AUTO, RENUM no puede generar números de línea más altos que 65529. Todos los GOTO y GOSUB que se encuentren en el programa son renumerados para tomar los nuevos números de línea. Si un GOTO o un GOSUB hacen referencia a una línea que no contiene ninguna sentencia BASIC, es decir, una línea que no exista, entonces la renumeración continúa, pero un error es generado por

cada aparición de un número de línea no existente. El mensaje generado es "línea *n* indefinida en *m*" (*Undefined Line* n *in* m), donde *n* es el número de línea que no existe y *m* es el número de línea donde está la sentencia que causó el problema.

RUN[3]

Teclear este comando originará, por el ordenador, la ejecución del programa BASIC residente actualmente en RAM, empezando por el número de línea más bajo del programa. Si la palabra RUN es seguida por un número de línea que pertenece al programa, el ordenador ejecutará el programa BASIC desde esa línea.

TRON

Este comando puede ser frecuentemente de un valor incalculable en el proceso de depurado de sus programas. Es abreviatura de *TRace ON*, y, después de ser activado, el programa que se está ejecutando imprimirá en la pantalla cada número de línea al tiempo que es ejecutada. Se anula mediante NEW o TROFF *(TRace OFF)*. Puede ser usado en una línea del programa o en modo directo.

```
10 PRINT "Hola"
20 PRINT "Adios"
30 TRON
40 PRINT "MSX"
50 TROFF
```

producirá lo siguiente cuando sea ejecutado:

```
Hola
Adios
[40] MSX
[50]
```

En el siguiente grupo de sentencias BASIC que consideraremos, todas ellas conciernen de alguna manera a variables.

CLEAR n, m

Ambos parámetros *m* y *n* son opcionales. Utilizando sólo CLEAR, el ordenador realizará las siguientes funciones:

- i Poner todas las variables numéricas a cero.
- ii Rellenar todas las cadenas a blancos.
- iii Cerrar cualquier fichero que estuviera abierto.

Si se especifica el parámetro *n*, el comando CLEAR *n* reserva el espacio para las variables de cadena. Cuando se conecta el ordenador, el BASIC MSX reserva 200 bytes de memoria para el uso de variables de cadena. Puede que usted requiera mayor espacio para sus variables; si eso ocurre, debe usar el comando CLEAR *n* para generar más espacio para ellas. CLEAR 250 reservará 250 bytes de RAM para variables de cadena. Para ver qué ocurre cuando usted opera sin espacio para éstas, teclee las siguientes instrucciones:

```
CLEAR 0
A$ = "Fede"
```

El parámetro *m* especifica la posición de memoria más alta accesible por las variables y programas BASIC. Esto permite al programador crear un área de RAM "fuera de fronteras" para el sistema BASIC. El área de RAM especificada de esta manera es un lugar seguro para que residan sus programas en código máquina.

ERASE "array1", "array2",...

Este comando le permite borrar selectivamente matrices[4] del espacio de variables del ordenador, sin afectar al valor de otras. El comando es muy útil para redimensionar matrices ya existentes, sin provocar el error matriz redimensionada *(Redimensioned Array)*. Utilice simplemente ERASE y redimensione las matrices usando sentencias DIM.

```
ERASE fr,ea
```

borrará las matrices fr y ea. El comando puede ser usado en líneas de programa o en modo directo.

INPUT

Es uno de los comandos más importantes del lenguaje BASIC. Hemos visto cómo podemos asignar valores a las variables en las líneas de un programa, de tal manera que, cuando se ejecuta el mismo, las variables toman los valores citados. Sin embargo, ¿qué ocurre si queremos cambiar el valor de la variable mientras el programa está rodando? Pues que el comando INPUT causa la interrupción del programa hasta que el usuario teclea los valores numéricos o de cadenas *(strings)* que serán asignados a ciertas variables específicas. La sintaxis para la instrucción INPUT es:

```
INPUT"cad. constante";lista variables
```

La cadena constante es opcional, y hablaremos ampliamente de ella más tarde. La lista de variables consta de un cierto número de nombres de variable separados por comas. Así:

```
INPUT A,S,D
```

permitirá al usuario asignar valores a las variables A, S y D. Observe cómo solamente requerimos el punto y coma cuando usamos la cadena constante. También se pueden asignar valores a las variables tipo matriz usando la sentencia INPUT, y las variables de cadena también pueden formar parte de la lista de variables. Así:

```
INPUT A$,A(1)
```

es perfectamente correcto. Cuando la instrucción arriba indicada aparece en un programa, el ordenador visualiza un *prompt*[5] en la pantalla en la forma de una interrogación. Si se especifica la cadena constante, como en

```
INPUT "Introduzca el primer numero";A
```

ésta es visualizada en la pantalla seguida de "?". La cadena constante se llama *prompt* de cadena *(prompt string).*

Cuando el usuario temlea una respuesta a la sentencia INPUT, el valor introducido debe encajar con el tipo de variable esperado por el ordenador según lo indicado en la sentencia INPUT. En

```
INPUT A(1),A(2),B
```

el ordenador espera que el usuario introduzca tres números; hay dos maneras de hacerlo. Si el usuario teclea un solo número y pulsa RETURN, un segundo *prompt*, "??", sería generado. Este tipo de *prompt* será generado cada vez que el ordenador espere otro grupo de datos. El segundo método es introducir todos los datos de una vez, separados por comas. Así, los datos precisos para la anterior sentencia INPUT podrían ser introducidos como se muestra seguidamente:

```
? 1,2,3 (RETURN)
```

Si se requiere la introducción de una cadena, no son necesarias las comillas. Si, no obstante, teclea una cadena cuando se espera un valor numérico, será emitido el mensaje de error "introduzca de nuevo los datos" *(Redo from Start)*, y usted tendrá que introducir de nuevo todas las respuestas en aquella sentencia INPUT, aunque los valores previamente introducidos fueran correctos. Si se introducen más datos de los esperados por la instrucción INPUT, el mensaje "ignorados los datos sobrantes" *(Extra Ignored)* será visualizado. Esto quiere decir exactamente lo que dice; los datos que fueron introducidos sobrepasando los requerimientos de la instrucción, sencillamente no son considerados.

Una instrucción INPUT puede ser interrumpida por CTRL-C o CTRL-STOP. El problema que presentan estos métodos es que el programa también se interrumpe. Aun así, puede hacerse continuar mediante el comando CONT.

LINE INPUT "cadena constante"; variable de cadena

Pruebe el siguiente programa:

```
10 INPUT A$
20 PRINT A$
30 GOTO 10
```

Ejecútelo y teclee unas cuantas cadenas para confirmar que funciona. Ahora introduzca una cadena que contenga una coma, y contemple lo que ocurre. Aparecerá el mensaje *Extra Ignored,* y la sentencia PRINT de la línea 20 solamente imprimirá la cadena de entrada antes de la coma. Una manera de evitarlo es expresar la cadena entre comillas, pero este método presenta problemas si quiere que aparezcan comillas como parte de la cadena. LINE INPUT es la solución a todos ellos. Cambie la línea 10 del ejemplo anterior por:

```
10 LINE INPUT A$
```

y vuelva a ejecutar el programa. Lo primero que notará es que el *prompt* "?" no aparece. Cada carácter que usted teclee es añadido a A$. Si desea que aparezca un *prompt* en la pantalla, puede usar una cadena constante, como se muestra a continuación:

```
LINE INPUT "Solo variables de cadena";a$
```

CTRL-C y CTRL-STOP tienen efectos similares a los que producen con INPUT.

READ, DATA y RESTORE

Imagine que hemos escrito un programa que requiere que los meses del año estén contenidos en una matriz de cadena llamada año$(12). Una manera de colocar los meses en la matriz es:

```
10 DIM ano$(12)
20 ano$(1)="Enero"
30 ano$(2)="Febrero"
 .
 .
 .
```

e igual con el resto de los meses. Esto funciona, pero ocupa mucho espacio de programa para hacer doce asignaciones de la matriz separadas. Una manera más

eficaz de asignar los meses a la matriz es utilizar sentencias DATA y READ, que proporcionan un medio para leer los meses del año de una lista contenida en el programa, y asignarlas a los elementos de la matriz. El bucle FOR-NEXT que hemos usado en este programa se explicará en este mismo capítulo. Por el momento, es suficiente decir que lo hemos utilizado para leer elementos de la lista de datos y asignarlos a los elementos particulares de la matriz; el valor del índice depende del valor de la variable I.

```
10 DIM ano$(12)
20 FOR I=1 TO 12
30 READ ano$(I)
40 NEXT I
50 END
60 DATA Enero,Febrero,Marzo,Abril,Mayo
70 DATA Junio,Julio,Agosto,Septiembre
80 DATA Octubre,Noviembre,Diciembre
```

Las líneas 60 a 80 proporcionan una lista de los meses que deseamos sean colocados en la matriz. Cada una comienza con la palabra DATA y consiste en una serie de constantes de cadena separadas entre ellas por comas. Desde luego, en otro programa la sentencia DATA puede ser una lista de números o una lista mixta, de números y cadenas combinados. No obstante, en cada caso los valores deben ser constantes, y deben estar separados unos de otros por comas. Si una constante de cadena de la lista contiene una coma como parte de ella, debe estar entrecomillada. También son necesarias las comillas si, como parte de la cadena, han de ser incluidos espacios encabezando o intercalados. Generalmente, estos espacios son descartados, al final o al principio de una cadena, cuando el ordenador hace uso de la lista. Las sentencias DATA no son ejecutadas por el programa. Una serie de estas sentencias, incluso aunque estén separadas por líneas con otras, se consideran como una larga lista de constantes en un programa. El comienzo de la lista se sitúa siempre en la sentencia DATA con el número de línea más bajo, y el final, al término de la DATA con el número de línea más alto.

Para acceder a las constantes contenidas en las instrucciones DATA empleamos una sentencia llamada READ. Esta puede ir seguida de un solo nombre de variable o de una lista de nombres de variable separados por comas. La lectura de las sentencias DATA se hace mediante la variable que sigue a la instrucción READ, siéndole asignado el siguiente valor de la sentencia DATA. Por ejemplo, si no se han emitido otros comandos READ, la primera sentencia READ asignará a su variable la primera constante de la sentencia DATA del programa. En el ejemplo anterior, la primera vez que se ejecutó el comando READ la constante "Enero" fue colocada en la matriz.

Para clarificar las ideas, imagine que hay un puntero[6] en el intérprete BASIC, que, cuando se ejecuta el programa, apunta al primer término de la primera sentencia DATA del mismo. Este valor se asignará a la variable de la primera sentencia READ

encontrada, y colocará el puntero en el siguiente término de la sentencia DATA. Cualquier operación de lectura subsiguiente colocará el puntero una posición más adelante hacia el final de la lista. Si el puntero está ya al final de una instrucción DATA, el siguiente comando READ provocará que el puntero se coloque en el primer término de la siguiente sentencia DATA. Si ésta no existe, el siguiente READ causará un mensaje de error: "sin datos" *(Out of Data)*. Si el puntero todavía señala un dato, pero no aparecen más instrucciones READ, los términos de datos sobrantes sencillamente se ignoran.

¿Qué ocurre si deseamos tomar datos de una sentencia DATA que ya ha sido leída? No podemos ir hacia atrás a lo largo de la lista usando READ, pero el comando RESTORE nos permite reinicializar el puntero al primer dato de una sentencia DATA en una línea particular del programa.

Cuando se usa solamente RESTORE se posicionará el puntero en el primer dato de la primera sentencia DATA del programa. Un comando como

```
RESTORE 3000
```

colocará el puntero señalando al primer dato de la sentencia DATA que haya en la línea 3000. No es posible colocar el puntero en un término dado dentro de una línea, solamente se puede en el primero.

MID$ (expresión de cadena 1,n,m) = expresión de cadena 2

Este comando nos ofrece la oportunidad de reemplazar parte de una cadena (la expresión de cadena 1) por otra (la expresión de cadena 2). La primera NO PUEDE ser una constante. *n* es la posición del primer carácter de la expresión 1 que será reemplazado por los caracteres de la expresión 2. Por ejemplo:

```
A$="QQQQQQQQ"
MID (A$,2)="ELP"
PRINT A$
```

retornará "QELPQQQQ" como valor de A$. *m* es opcional y representa el número de caracteres de la cadena 2 que se han de trasladar a la cadena 1. El resultado de la función será una cadena nunca más larga que la cadena colocada en primer lugar.

SWAP[7] var1, var2

Este comando, como su nombre implica, intercambia los valores de las variables 1 y 2. Dichas variables pueden ser numéricas o de cadena, pero no es posible intercambiar los valores entre una variable de cadena y otra numérica. Ninguno de los parámetros del comando SWAP puede ser una constante.

* * *

El siguiente grupo que veremos son comandos BASIC que no se refieren a un dispositivo específico. Estos comandos, que trataremos en capítulos posteriores, trabajan enviando o leyendo información de otros *chips* en el ordenador MSX; estrictamente hablando, se puede decir que comandos como PRINT e INPUT son de función específica, ya que obviamente envían datos al VDP para que aparezcan en la pantalla. No obstante, estos dos comandos están disponibles en todas las versiones del lenguaje BASIC, y por eso está justificado decir que no son comandos de función específica. Ejemplo de esto último son el comando SCREE*n*, que opera conjuntamente con el VDP, y SOUND, que trabaja con el PSG.

DEF USRn = expresión entera

Lo utilizan programadores avanzados para informar al ordenador de la dirección de comienzo de una rutina en código máquina. *n* es un dígito entre 0 y 9, y si se omite, el valor asumido por el ordenador es 0. La expresión entera es la dirección de comienzo de la rutina en código máquina, y debe ser un entero entre 0 y 65535. Las direcciones correspondientes a los diferentes valores de *n* pueden ser redefinidas a lo largo del programa tantas veces como sea necesario. El comando USR*n* se usa conjuntamente con éste, y en su apartado correspondiente se darán detalles más concretos acerca de él.

END

Es un comando muy simple. Se utiliza para cesar la ejecución del programa, y hace que el ordenador regrese a modo directo.

ERROR expresión entera

Como sin duda habrá advertido, su ordenador le comunica cuándo ha cometido un error en su programa. Todos los mensajes de error que puede emitir el ordenador tienen un número asociado. Así, el número asociado al mensaje de error *Out of Data* es 4. El comando ERROR nos permite simular cualquier error usando su número de código. Por ejemplo, si emitimos la instrucción

```
ERROR 4
```

provocará que sea impreso el mensaje *Out of Data* exactamente igual que si el error se hubiera producido por un intento de lectura incorrecta de datos. Si la sentencia aparece en un programa en ejecución, éste se interrumpirá al tiempo de emitir el mensaje. Sin embargo, los errores generados de esta manera pueden ser tratados, como todos ellos, con el comando ON ERROR, que veremos en el capítulo 5. Si experimenta con la instrucción ERROR, encontrará que, para algunos valores de la expre-

sión entera, se visualiza el mensaje "error no imprimible" *(unprintable error)*. Tales números se pueden usar para generar errores "definidos por el usuario", originando el pase del control a la rutina ON ERROR, si la contiene el programa, o que se interrumpa con el correspondiente mensaje de error, en caso contrario. Los números que dan este error son 23, 26 al 49, y 60 al 255. Si usted quiere añadir sus propios mensajes de error, tenga en cuenta que para hacerlo debe usar los números entre 60 y 255, ya que los otros están reservados para futuras ampliaciones del sistema MSX.

ERR y ERL

Son variables del sistema, es decir, variables cuyo valor sólo puede ser alterado por el BASIC del sistema MSX por el código máquina o por algún otro método de programación avanzada.

ERR contiene el número del último error ocurrido, si éste fue causado por un error de programa o por el uso de la sentencia ERROR. Se hablará más acerca del tema cuando tratemos ON ERROR en el capítulo 5.

ERL proporciona el número de línea donde se produjo el último error, pero no es responsable de la causa que lo originó. Si el último error se produjo en modo directo, ERL toma el valor 65535.

FOR var = x TO y STEP z
NEXT var

Esta es la primera ESTRUCTURA DE CONTROL que encontramos en el BASIC MSX. Una estructura de control es una instrucción que gobierna el flujo de programa. Normalmente, un programa empieza en el número de línea más bajo y prosigue la ejecución siguiendo el orden de éstas. No obstante, si necesitamos ejecutar algunas líneas repetidamente, o saltarlas, podemos usar lo que se denomina una estructura de control. El bucle FOR-NEXT, como se llama la estructura que ahora tratamos, nos permite ejecutar un bloque de líneas de programa un número dado de veces. Otras estructuras de control son GOTO, IF ... THEN ... ELSE, y GOSUB ... RETURN, que también examinaremos en este capítulo.

x es el valor inicial de la variable *var*, que se llama variable de control del bucle. *y* es el valor final, o valor límite, que tomará *var*. *z* es opcional, y *x*, *y*, *z* son todas ellas expresiones numéricas.

Las líneas de programa entre FOR y su correspondiente NEXT son ejecutadas repetidamente, hasta que el valor de *var* excede al valor límite.

Después de que las sentencias entre FOR y NEXT han sido ejecutadas, la variable *var* es incrementada en uno o, según el valor de z si la palabra STEP está presente. Una vez que el valor de *var* excede al valor límite, la sentencia que sigue inmediatamente a NEXT es ejecutada. El bucle

```
10 FOR I=1 TO 10
20 PRINT I
30 NEXT I
```

imprimirá los números del 1 al 10 en la pantalla. Si hubiéramos puesto STEP 2 en la sentencia FOR, sólo habrían sido impresos los números 1, 3, 5, ... Experimente con el comando para acostumbrarse a él. Si quiere ir desde un valor alto de *x* hasta uno bajo de *y*, simplemente dé un valor negativo a STEP. Los valores de *x*, e *y*, *z* no han de ser necesariamente enteros, pero en muchas aplicaciones de programación sí lo son. El nombre de variable que sigue a la sentencia NEXT no es obligatorio, pero, si está presente, ayuda a la legibilidad del programa.

Podemos tener bucles FOR-NEXT unos dentro de otros. A esto se le llama ANIDAR bucles. En estos casos de bucles anidados es esencial que el FOR del bucle interior encuentre su correspondiente NEXT antes que el del bucle exterior. Así:

```
FOR I=1 TO 10
FOR J=1 TO 10
.....
NEXT J
NEXT I
```

es legal, mientras que

```
FOR I=1 TO 10
FOR J=1 TO 10
.....
NEXT I
NEXT J
```

causará problemas. El tipo de variables numéricas tomado por defecto en los bucles FOR-NEXT es "doble precisión". Muy raramente se necesita tanta exactitud en esta aplicación, y por ello pueden usarse en su lugar variables de tipo entero o de simple precisión. Esto tiene dos efectos: se ahorra espacio al almacenar las variables en memoria, y los bucles se ejecutan mucho más rápidamente cuando se usan variables de tipo entero o de simple precisión. Pruebe el programa siguiente, con variables de tipo "entero", "simple" y "doble precisión" para I. Más adelante daremos detalles acerca de la función TIME. Aquí sólo la usamos para obtener un tiempo de respuesta relativo para cada tipo de variable.

```
10 DEFDBL I
20 TIME = 0
30 FOR I=1 TO 200: NEXT I
40 PRINT TIME
```

Los resultados obtenidos en el ordenador Sony HB-55 MSX[8] son los siguientes:

Tipo de I	Tiempo
Doble precisión	23
Simple precisión	20
Entero	10

Si se omite la I de la sentencia NEXT los tiempos son 19, 16 y 6, respectivamente. Poniendo el NEXT en una línea separada tenemos 20, 17 y 7. Por eso, si se necesita velocidad, la variable de control de un bucle FOR-NEXT debe ser entera.

GOSUB y RETURN

Frecuentemente tenemos secuencias de sentencias que se repiten en varios lugares a lo largo de un programa. Podemos reemplazar cada una de esas secuencias por una instrucción GOSUB *n*, donde *n* es el número de línea de la secuencia de instrucciones que queremos ejecutar en ese punto del programa. De este modo solamente necesitamos una copia del conjunto de instrucciones que ha de ser incluido en el programa, y a esta copia se la llama SUBRUTINA. Una subrutina siempre acaba con el comando RETURN, que devuelve el control del programa a la sentencia siguiente al GOSUB. El número de línea referenciado por la sentencia GOSUB debe ser una constante numérica; a diferencia de otras versiones BASIC, el número de línea no debe ser una variable o una expresión.

Al escribir un programa es una buena idea separar sus subrutinas de la parte principal del mismo mediante un comando END o STOP. Si se ejecuta un RETURN sin su correspondiente GOSUB, se genera un error. Por esta razón, tendemos a mantener todas las definiciones de subrutinas al final de nuestros programas, y las dividimos mediante sentencias REM (ver más adelante) para indicar el nombre y la función de la subrutina siguiente.

GOTO[9] n

Como ya hemos apuntado, un programa se ejecuta generalmente siguiendo el orden numérico de sus líneas. Una sentencia GOTO *n* propiciará que el control del programa pase a la línea *n*. Uselas cuidadosamente: un programa lleno de sentencias GOTO es muy difícil de leer y comprender. Como ocurría con las sentencias GOSUB, el número de línea especificado debe ser una constante. Si este número de línea no existe, aparecerá un error. El comando puede usarse en modo directo con el propósito de rodar un programa —o una porción del él— sin inicializar las variables, tecleando GOTO *n*, siendo *n* el número donde desee iniciar la ejecución del programa.

IF expresión THEN sentencias ELSE sentencias
IF expresión THEN GOTO nn ELSE sentencias

Esta estructura de control permite al ordenador ejecutar ciertas sentencias solamente si se dan unas condiciones preestablecidas. La expresión es una BASIC que retorna un resultado "verdadero" o "falso". Si al evaluar la expresión el resultado es verdadero, se ejecutan las sentencias inmediatamente después de THEN. Si se da el caso contrario, las que se ejecutan son las sentencias que siguen a ELSE. De esta manera, solamente se ejecuta uno de los dos grupos de sentencias cuando el programa llega a esa línea. La construcción IF ... GOTO es un caso especial de la IF ... THEN, donde no es necesario THEN. En este caso, se tiene en cuenta GOTO *nn* solamente si la expresión evaluada es cierta. En la sentencia IF ... THEN ... ELSE, las instrucciones precedidas por THEN y ELSE pueden ser sustituidas, si se desea, por números de línea.

```
100 IF I<6 THEN 200 ELSE 300
```

que es el equivalente de la sentencia.

```
100 IF I<6 THEN GOTO 200 ELSE GOTO 300
```

Las sentencias IF ... THEN ... ELSE pueden anidarse, acompañando cada ELSE al THEN desapareado más cercano. Por otra parte, esto puede crear bastante confusión, y, siempre que sea posible, es aconsejable mantener estas sentencias en líneas separadas. Si el número de línea que sigue a ELSE, THEN o GOTO no existe, se generará un error.

KEY, KEY LIST

Una característica interesante de los ordenadores MSX es que tienen una serie de teclas llamadas "teclas de función". Probablemente usted conocerá ya que varias palabras BASIC pueden ser introducidas en un programa o en modo directo simplemente pulsando la tecla apropiada. Por ejemplo, pulsando F2 se activa el comando AUTO, y F5 hará rodar el programa BASIC residente actualmente en su máquina. Sin embargo, también es posible modificar lo que hacen estas teclas usando el comando KEY *n*, donde *n* es un número entre uno y diez. Así, los comandos

```
A$="PRINT"
KEY 1,A$
```

motivarán que la palabra PRINT sea escrita cada vez que se pulse KEY 1. En este ejemplo, el cursor quedará inmediatamente después de la palabra, de tal manera que usted puede insertar más texto. Si desea que la sentencia que activa mediante una tecla de función sea ejecutada inmediatamente, puede hacer que el ordenador

piense que ha sido pulsado RETURN cuando se presiona la tecla de función según el método siguiente (CHR$(13) simula que se ha pulsado la tecla RETURN):

```
KEY 1,"PRINT A"+CHR$(13)
```

La cadena que activemos en la tecla de función ha de ser de 15 caracteres como máximo.

El comando KEY LIST presenta el contenido actual de todas las teclas de función.

En el capítulo 6 se efectuará un estudio más profundo de cómo pueden utilizarse las teclas de función, cuando examinemos el comando ON KEY.

ON GOTO y ON GOSUB

Aunque todos estos comandos empiecen con la palabra ON, son ligeramente diferentes en su funcionamiento. Los comandos ON se exponen en el capítulo 5. Estos comandos posibilitan transferir el control a un número de línea dependiendo del valor de una expresión o variable BASIC. La sintaxis se muestra a continuación:

```
ON expresion GOTO linea1,linea2,linea3...
```

Si, por ejemplo, el valor de la expresión es 3, entonces saltará la ejecución del programa al tercer número de línea en la lista de números después del GOTO. Así, en la sección del programa

```
100 A=2
110 ON A GOTO 200,300,400
```

el número de línea de destino sería 300. Si la expresión evaluada no es un número entero, simplemente no es considerada la parte fraccionaria. En la construcción ON ... GOSUB, los números de línea son los primeros de las subrutinas.

Si el valor retornado es 0 —o más alto que el número de términos en la lista de números de línea, pero menor que 255—, la ejecución del programa continúa en la sentencia siguiente a la construcción ON ... GOTO u ON ... GOSUB. Si, no obstante, la expresión retornada resulta ser un número negativo o mayor que 255, se emitirá un error denominado "llamada a una función ilegal" *(illegal function call)*.

POKE dirección, expresión entera

Este comando se usa cuando necesitamos alterar directamente un valor contenido en una cierta posición de RAM. La dirección en la sintaxis expresada anteriormente es la del byte a alterar, y la expresión entera es el nuevo valor que tomará la posi-

ción de memoria. La dirección debe estar comprendida entre −32768 y +65535. Si el valor de la dirección es negativo, la máquina añadirá 65535 a la dirección para obtener la posición del byte a alterar.

PRINT lista de expresiones

Ya hemos usado el comando PRINT[10] en un par de programas de demostración. Como su nombre indica, imprime el valor de una expresión en la pantalla. Cuando se emite el comando PRINT aislado, ya sea desde el modo directo o como parte de una línea de programa, se imprimirá una línea en blanco en la pantalla. PRINT, seguido de una expresión numérica o de cadena, visualizará el valor de la expresión en el *display*. El BASIC MSX divide cada línea de la pantalla en ZONAS DE IMPRESION. (**Nota:** El comando PRINT no funcionará con ciertos modos de pantalla; esto se tratará con más detalle cuando examinemos el VDP.) Cada zona de impresión tiene 14 caracteres de largo. Dónde se imprime un valor en relación con estas zonas depende del carácter usado para separar las expresiones en la lista de parámetros del comando.

"," Provoca que el valor de la siguiente expresión se imprima al principio de la zona de impresión siguiente.

";" Provoca que la siguiente expresión sea impresa inmediatamente después de la última.

Si una sentencia PRINT termina con alguno de estos caracteres, llamados frecuentemente DELIMITADORES, la siguiente PRINT visualizará sus expresiones de acuerdo con la anterior. Si la lista de expresiones es demasiado larga como para que quepa en una línea, o si una sola expresión retorna un valor cuya longitud excede a la de una línea, los caracteres sobrantes serán impresos en la siguiente línea. El carácter "?" puede usarse en lugar de la palabra PRINT como en

```
? "Hola"
```

que equivale a PRINT "Hola".

PRINT es uno de los comandos con más posibilidades que nos hemos encontrado hasta ahora. A causa de la variedad de maneras de imprimir en las zonas de impresión de una línea dada, bien merece la pena estudiar más el comando, viendo exactamente qué podemos hacer con él.

PRINT USING expresión de cadena; lista de expresiones

El comando PRINT USING nos facilita la impresión de cadenas o números en el *display* de acuerdo con un formato prefijado. Por ejemplo, nos permite imprimir expresiones numéricas estableciendo el número de dígitos que han de aparecer delante y detrás del punto decimal, muy útil si tenemos que imprimir tablas de datos nu-

méricos en la pantalla. La expresión de cadena de la sintaxis anterior se llama cadena de formato, y contiene ciertos caracteres no alfanuméricos.

Echemos una mirada a estos caracteres de formato, examinando primero los que se usan para formatear expresiones de cadena.

"!"

Este carácter imprime sólo el primero de cada cadena de la lista de expresiones; por ejemplo:

```
PRINT USING "!";"Hola";"Adios"
```

retornará

```
HA
```

como resultado. La incorporación de expresiones numéricas en la lista de expresiones genera un error tipo de variable incorrecta *(type Mismatch).*

***n* espacios** &[11]

Este formato de cadena consta de dos & separados por *n* espacios. 2+*n* caracteres de la cadena en la lista de expresiones serán impresos en la pantalla. Si pone *n* igual a 0, entonces serán impresos dos caracteres de la cadena. Si 2+*n* es más largo que la cadena, la expresión de cadena se imprime en el *display* con los espacios necesarios al final.

```
PRINT USING "& &";"GANADO"
```

imprimirá "GAN" en el *display.*

@[12]

Este carácter nos da un método para colocar variables de cadena en la mitad de una constante de cadena. Es un método ligeramente diferente de los otros dos caracteres de formato que hemos usado hasta ahora, y realmente no hay un formato de la cadena como en:

```
A$="MSX"
PRINT USING "Este es un ordenador @";A$
```

Para formatear expresiones numéricas se usan otros caracteres. Veamos ahora esto de una manera similar.

#

Este carácter nos permite especificar el número de dígitos que deseamos imprimir antes y después del punto decimal en una expresión numérica. Por ejemplo, "# # . # #" como cadena de formato especificará que se impriman dos dígitos antes del punto decimal y dos después.

```
PRINT USING "##.##";10.2
```

presentará

```
10.20
```

en el *display*. Reemplace el 10.2 por 100.7. Este será impreso como %100.70. El "%" indica que hay un exceso de un dígito en el número (en este caso, el 1 en la columna de las centenas). Pruebe con otros números, como 0.37 o 1.37. En tales casos se imprime un espacio a la izquierda del primer dígito. Un signo "+" al principio o al final de la cadena de formato imprimirá el signo del número al principio o al final del mismo. Por ejemplo:

```
PRINT USING "+##.##";1.3
```

imprimirá el signo del número a su izquierda.

Un carácter "−" al final de la cadena formato imprimirá el número con el número de dígitos apropiado, y con un signo menos (−) al final del mismo en caso de que tuviera que llevarlo.

Si se añaden asteriscos "**" al frente de la cadena de formato para indicar el número de dígitos a imprimir, el número aparecerá en la pantalla con asteriscos a la izquierda en lugar de espacios si faltan dígitos.

```
PRINT USING "**.##";2.2
```

imprimirá en el *display* * 2.20. De manera similar

```
PRINT USING "**#.##";10.3
```

imprimirá * 10.30. Dése cuenta de que los números con este formato son redondeados por exceso o por defecto de acuerdo con su valor, de modo que el número es exacto con el número de dígitos mostrado.

¥¥

Es de uso mínimo en Europa. Se ha de usar junto con el carácter "#", y representa dos posiciones de dígitos a la izquierda del punto decimal, siendo ocupada una de las posiciones por un carácter "¥". Así:

```
PRINT USING "¥¥.##";1.3
```

imprimirá en la pantalla ¥1.30.

“,”

La coma es bastante útil en el formato numérico. De nuevo es usada conjuntamente con el carácter “#”. Colocada a la izquierda del punto decimal, hace que se imprima una coma a la izquierda de cada tres dígitos de la parte entera del número. Por ejemplo:

```
PRINT USING "####,.##";10000.2
```

se visualizará, 10,000.20.

“∧ ∧ ∧ ∧”

Colocado tras los caracteres “#” en una cadena de formato numérico, indica que los números han de presentarse en formato exponencial.

```
PRINT USING "#.#^^^^";7.4
```

imprimirá 0.7E+01. Puede producirse un error si especificamos más de veinticuatro dígitos en una cadena de formato. Por otra parte, una cadena de formato puede ser una variable de cadena. Ejemplo: A$=“##·##”.

REM

Abreviatura de REMark (comentario), le permite introducir comentarios en el programa sin que tengan efecto sobre su correcta ejecución.

```
100 REM Esto es un comentario
```

La línea puede ser direccionada por un comando GOTO o GOSUB. La ejecución continúa en la siguiente sentencia posterior a REM. Cualquier sentencia en la misma línea de REM, separada de ella por “:”, es ignorada. El carácter “'” puede usarse al final de una línea en lugar de REM.

STOP

Este comando interrumpe la ejecución del programa BASIC que estuviera funcionando en ese momento. El control es pasado al modo directo. A diferencia de

END, no provoca el cierre de los ficheros que estuvieran abiertos. La ejecución puede proseguir con CONT.

Funciones intrínsecas

Ya hemos considerado las sentencias y comandos BASIC. Una función es una serie de operaciones que se realizan sobre las variables y constantes BASIC. Se darán más detalles en el siguiente capítulo, pero aquí vamos a estudiar las funciones intrínsecas del sistema. Estas funciones están presentes en los ordenadores MSX desde el instante en que se conectan. Los programas que trabajan con las funciones están almacenados en ROM como parte del intérprete BASIC. Veámoslas ahora en orden alfabético.

ABS(n)

Esta función da el valor ABSOLUTO del número *n*. Es el valor del número despreciando su signo. Así, ABS(−10) es 10, y evidentemente es mayor que ABS(5).

ASC(n$)

Devuelve el código ASCII de *n$*.

ATN(n)

Retorna el arco-tangente de *n* en radianes. Todas las funciones trigonométricas, tales como SIN, TAN, COS, trabajan en radianes y en doble precisión. El resultado de esta función siempre está comprendido entre −PI/2 y +PI/2.

BIN$(n)

El resultado es una cadena que representa el valor binario de *n*. *n* es una expresión numérica con un valor entre −32768 y +65535. Si *n* es un número negativo, la cadena retornada se representa en forma de complemento a 2 del número.

CDBL(n)

Convierte *n* a doble precisión.

CHR$(n)

Devuelve el carácter ASCII que tiene el número *n*.

CINT(n)

Genera la parte entera truncada de *n*. *n* debe estar entre −32768 y 32767.

COS(n)

Proporciona el coseno de *n* en radianes.

CSNG(n)

Convierte *n* a simple precisión.

CSRLIN

No tiene argumento, pero retorna la coordenada vertical actual del cursor en la pantalla. En el capítulo del VDP veremos las leyes que rigen la pantalla.

EXP(n)

Da *e* elevado a *n*. *n* debe ser menor que 145.1, o aparecerá un error de desbordamiento *(overflow)*.

FIX(n)

Retorna la forma entera de *n*, pero cuando se usa con números negativos no retorna el siguiente número negativo más bajo, como hace CINT.

FRE(0)

El argumento aquí no tiene importancia; PRINT FRE(0) retornará el número de bytes libres para su programa, etc.

FRE(" ")

Indica el número de bytes libres para cadenas de caracteres.

HEX$(n)

Devuelve una cadena representando el valor hexadecimal de *n*. Trabaja de manera similar a BIN$.

INKEY$

Retorna una cadena de un carácter que representa la tecla pulsada, o una cadena vacía si no se pulsa ninguna. La tecla pulsada no tiene eco en la pantalla, es decir, no es impresa en pantalla cuando se pulsa.

```
10 A$=INKEY$
20 IF A$="" THEN GOTO 10
30 PRINT ASC(A$)
40 GOTO 10
```

Este programa imprime el código ASCII de las teclas pulsadas. Aprecie que, si presiona una tecla de función, el programa presentará todos los códigos ASCII de la cadena generada.

INPUT$(n)

Esta función acepta *n* caracteres antes de permitir que el ordenador continúe la ejecución del programa. INPUT$(1) aceptará un carácter; los caracteres así aceptados tampoco tienen eco en pantalla. Esta función es bastante útil, ya que nos permite escribir subrutinas que responden solamente si se pulsan ciertas teclas. Por ejemplo, la siguiente rutina espera que pulse la barra de espaciado antes de proseguir.

```
1000 REM Rutina para la barra espaciado
1010 PRINT "Pulse la barra para seguir"
1020 G$=INPUT$(1)
1030 IF G$<>" " THEN GOTO 1020
1040 RETURN
```

INSTR(n, x$, z$)

Esta función busca en qué lugar de la cadena x$ aparece la z$. Si *n* está presente, se busca desde el carácter *n* de x$ en adelante. Si no, se explora la totalidad de x$. *n* debe estar entre 0 y 255. La función retorna un 0 si no aparece z$ o si ambas cadenas son nulas. Si se encuentra z$, la función devuelve la posición en la que se encuentra z$ dentro de x$.

INT(n)

Retorna la parte entera de *n* con simplemente descartar la parte fraccionaria.

LEFT$(x$, n)

Nos devuelve los *n* primeros caracteres de x$. Por ejemplo:

```
PRINT LEFT$("ADIOS",3)
ADI
```

LEN(a$)

Retorna el número de caracteres de la cadena a$. Incluye los caracteres no imprimibles y los espacios.

LOG(n)

Da como resultado el logaritmo natural de *n*. *n* debe ser mayor que 0.

LPOS(0)

Se usa solamente con impresora. Devuelve la posición actual de la cabeza de la impresora.

MID$(a$, n, m)

Retorna una cadena de *m* caracteres de a$ empezando en el carácter *n*. *m* se puede omitir, y entonces la función dará todos los caracteres que están a la derecha del carácter *n*.

OCT$(n)

Es similar a BIN$(*n*), pero representa el valor octal de *n*.

PEEK(n)

Devuelve el valor del byte asociado a la dirección de memoria *n*. *n* debe estar comprendido entre −32768 y +65535. Vea POKE si necesita más detalles.

POS(0)

Retorna la posición horizontal del cursor de la pantalla en ese momento. El argumento es intrascendente. La columna 0 es la situada más a la izquierda de la pantalla.

RIGHT$(a$, n)

Devuelve los *n* últimos caracteres de a$.

RND(n)

Genera un número aleatorio entre 0 y 1. Si $n = 0$ el número generado es el último que generó la función. Si Si $n > 0$, el generador de números aleatorios los produce verdaderos. Si deseamos que estén entre 0 y 10, por ejemplo, podemos usar una función definida por el usuario:

```
10 DEF FNr(Q)=INT (RND(1)*11)
```

Observe cómo se usa 11 como multiplicador. Esto es porque la función INT siempre redondea por exceso; por ello, si hubiéramos usado 10, este número nunca se habría dado.

SGN(n)

Nos da un número que representa el signo de *n*. Si $n = 0$, la función retorna cero. Si $n < 0$, retorna -1, y si $n > 0$, entonces $+1$.

SIN(n)

Devuelve el seno de *n*.

SPACES(n)

Retorna una cadena de *n* espacios. *n* debe estar comprendido entre 0 y 255.

SPC(n)

Es similar a SPACES(*n*), pero sólo puede usarse con las sentencias PRINT o LPRINT.

SQR(n)

Retorna la raíz cuadrada de *n*.

STRING$

Esta función devuelve una cadena de caracteres de una manera similar a SPACES(*n*). STRING$(*n*, *m*) retornará una cadena de *n* caracteres cuyo código ASCII es *m*. STRING$(*n*,a$) dará una cadena de *n* caracteres, siendo el carácter a repetir el primer de a$.

TAB(n)

Indica la posición horizontal *n* de la pantalla donde comenzará a imprimir la siguiente sentencia PRINT. Si la posición de impresión ha pasado ya de la posición *n*, la función es ignorada. Este comando se usa conjuntamente con PRINT o LPRINT.

TAN(n)

Retorna la tangente de *n*.

TIME

Esta variable del sistema nos da acceso a un reloj interno de los ordenadores MSX. Como hemos mencionado previamente, la variable TIME se incrementa 50 veces por segundo en los ordenadores MSX equipados con monitor de TV PAL (por ejemplo, los modelos españoles e ingleses), y 60 veces por segundo en los ordenadores MSX con sistema de *display* NTSC. Si ponemos TIME a cero se reinicializa el reloj. TIME no se incrementa durante las operaciones con cinta, pero retiene el valor que tenía antes de que comenzara la operación.

USRn(m)

Es usado para llamar a una rutina en código máquina, ya sea en ROM o una realizada por el usuario. *n* es un dígito entre 0 y 9, e indica al ordenador la dirección de la rutina buscada. La dirección tiene que haber sido asignada previamente a USR*n* por una sentencia DEF USR. *m* es el argumento de la función y será dado como parámetro a la rutina en código máquina.

VAL(a$)

Retorna el valor numérico de a$. Por ejemplo:

```
A$="12"
PRINT VAL(A$)
12
```

VARPTR

Es una función de gran ayuda para los programadores avanzados, y ofrece un método para buscar la posición de memoria donde están almacenadas las variables. PRINT VARPTR(*n*) retornará la dirección del primer byte asociado a la variable *n*. Si *n* no ha sido asignada, se genera un error. La dirección retornada será mayor que −32768 y menor que 32767. Si la dirección es negativa, simplemente sume 65536

```
PRINT VARPTR(a(0))
```

devolverá la dirección de comienzo en RAM del primer byte del elemento 0 de la matriz *a*. Obsérvese que la dirección de la matriz en memoria es tratada como cualquier otra variable asignada. Por tanto, siempre que necesite esta información, utilice de nuevo VARPTR.

VARPTR (# número de fichero) retorna la dirección del primer byte del bloque de control de fichero. Es el único valor que necesita si quiere acceder directamente al bloque de control de fichero. ¡Tenga mucho cuidado al hacer esto, ya que es posible que confunda completamente al sistema de cinta MSX!

Aquí acaban las sentencias BASIC simples que son accesibles al programador de BASIC MSX. En el próximo capítulo veremos con más detalle todo este material y cómo trabajan las sentencias y funciones, así como las variables, constantes y expresiones.

NOTAS DEL CAPITULO

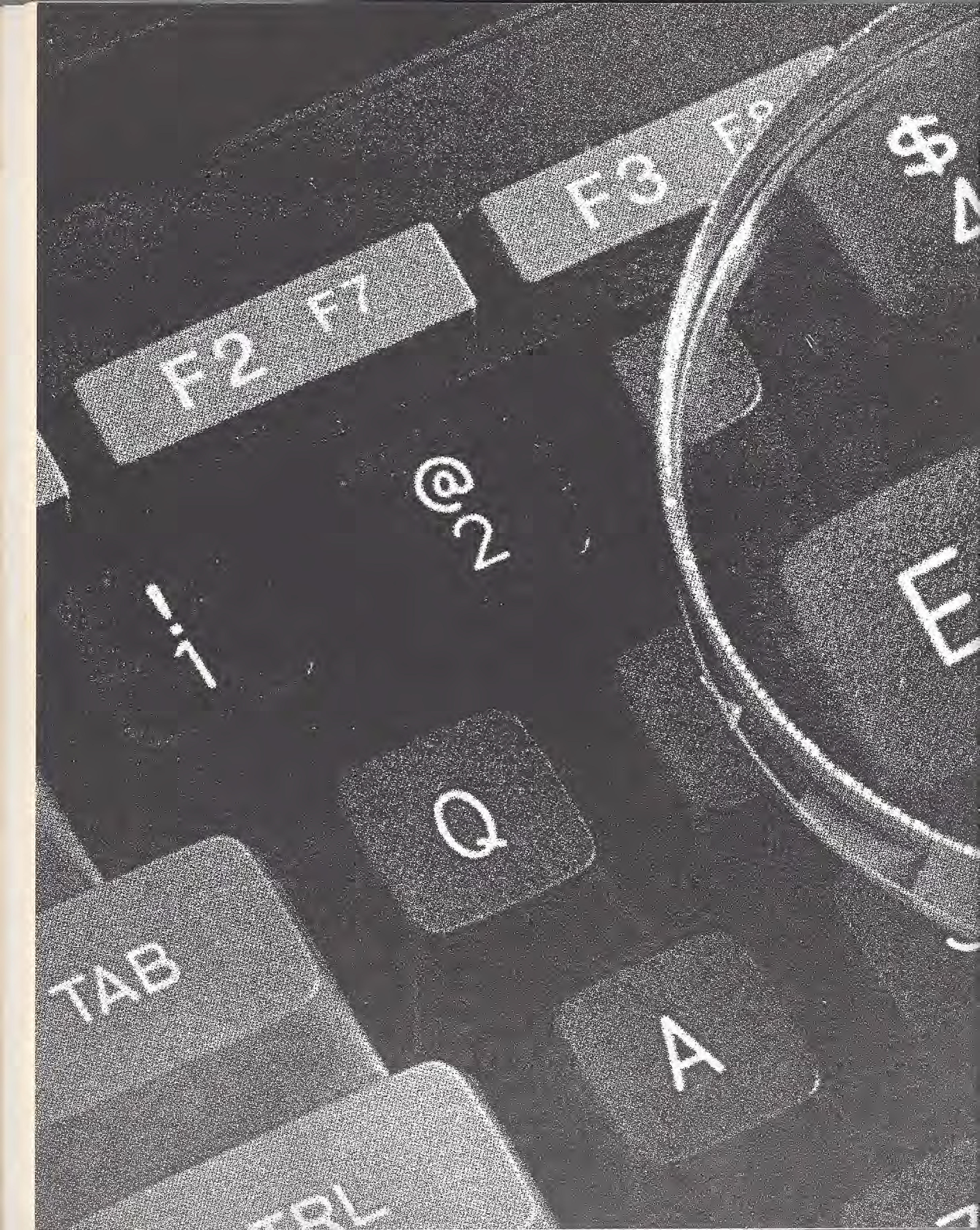
1.—RETURN: Tecla que se usa para introducir datos, programas, etc. al ordenador. En algunos ordenadores se llama ENTER o CR (*carriage return* o retorno de carro).
2.—LIST *n-m*: Se produce el mismo efecto pulsando la tecla de función F4.
3.—RUM: De la misma manera, el programa es ejecutado al pulsar la tecla de función F5.
4.—ARRAY: La traducción de *array* es "tabla o matriz de datos".
5.—PROMPT: Carácter o cadena de caracteres que aparecen en la pantalla para indicar al usuario que el ordenador espera una introducción de datos o comandos. Cuando se trata del carácter "■", intermitente o no, se le suele llamar cursor.
6.—Dirección que apunta a una palabra o grupo de palabras, en una tabla, en una fila de espera, una cadena, una lista o una pila. Esta última es muy utilizada.
7.—En inglés significa "intercambiar".
8.—Con el ordenador Spectravideo 728 MSX hemos obtenido los resultados siguientes:

Tipo de I	**Tiempo**
Doble precisión	19
Simple precisión	17
Entero	9

Omitiendo la I de la sentencia NEXT, los tiempos son 16, 13 y 5, respectivamente. Si ponemos el NEXT en una línea separada, los tiempos son:

poniendo la I: 20, 17 y 9, respectivamente;
sin poner la I: 16, 14 y 6, respectivamente.

9.—La palabra GOTO se puede conseguir pulsando la tecla F3.
10.—En inglés significa "imprimir".
11.—Según el manual del ordenador Spectravideo 728 MSX, el carácter "&" indica campo de longitud variable, y para indicar un formato de longitud fija se usa "/*n* espacios/".
12.—Se ha comprobado que, en el ordenador anteriormente mencionado, no funciona este tipo de formateo, ni aparece en su manual.


F2 F7
F3
@
2
!
1
Q
TAB
A
E
$



3 Estructuras de datos y variables

Todo programa de ordenador procesa información de una manera u otra. La información sobre la que actúa puede ser el nombre y la dirección de alguien, el número de días del año, la posición de un invasor del espacio en un videojuego, o cualquier otra cosa. Sin embargo, antes de que un ordenador pueda procesar los datos, éstos deben ser representados en el ordenador de una manera apropiada. Las formas de almacenar datos en el ordenador se llaman ESTRUCTURAS DE DATOS. Veámoslo con más detalle. El lector que esté interesado en los sistemas de numeración en general, puede consultar el apéndice. La primera estructura de datos que vamos a ver es el carácter, porque, dentro del ordenador, un carácter se puede representar por un solo byte de memoria.

Caracteres

¿Cómo puede trabajar un ordenador con datos no numéricos si, después de todo, es una máquina principalmente numérica? Los datos de texto, como la letra "A", se almacenan en el ordenador como un número entre 0 y 255. Esta cantidad, como ya habrá observado, se ajusta a un byte. Cada carácter del teclado del ordenador tiene un código numérico asociado, y el método más común de codificar caracteres es usar el código ASCII[1]. ASCII es el código estándar americano para el intercambio de información y son las siglas de *American Standar Code for Information Interchange.* Este código es el que utilizan las máquinas MSX, y en él la letra "A" se representa por

el número 65. Una "a" tiene asociado el código 97. Otros caracteres tienen diferentes códigos ASCII. Para examinar estos códigos asociados a los diferentes caracteres podemos usar una función BASIC llamada ASC( ). Tecleando

```
PRINT ASC("B")
```

y pulsando RETURN nos dará el resultado 66. Hay una función que realiza la operación contraria; dado el código ASCII de un carácter, esta función, CHR$( ), imprimirá el carácter. Por tanto,

```
PRINT CHR$(66)
```

imprimirá la letra "B". Algunos caracteres, tales como el carácter de código 1, no imprimirán nada en la pantalla; se les conoce como "caracteres NO IMPRIMIBLES". Otros caracteres se comportan de manera imprevista cuando se imprimen (intente imprimir el carácter 7 y el 12).

Cadenas

Este libro está hecho de cadenas. Una cadena (o *string*) es un conjunto de caracteres, y el carácter 13 es el que marca el final de una cadena.

Constantes

Una constante en un programa es un valor que no cambia durante la ejecución del mismo. Puede haber constantes numéricas o de cadena; por ejemplo, 1.234 y "Hola". Una cadena constante puede ser hasta de 255 caracteres. Existen seis maneras de representar las constantes numéricas en los ordenadores MSX, que ahora veremos. Una constante numérica puede ser positiva o negativa.

Constantes enteras

Una constante entera puede tener un valor entre −32768 y +32767 en el BASIC MSX. Obviamente, las constantes enteras no contienen punto decimal.

Constantes de punto fijo

Son números que contienen punto decimal.

Constantes de punto flotante

Son números positivos o negativos representados en formato exponencial. Por ejemplo:

1.234 E + *n*

Aquí, *n* es el exponente, y el número que está a la izquierda de E es la mantisa. Las constantes de punto flotante han de estar comprendidas en el rango de 10E-64 y 10E+63.

Constantes hexadecimales

Son números hexadecimales, y están precedidos por los caracteres &H.

Constantes octales

Las constantes octales han de estar precedidas por &O o sólo por &.

Constantes binarias

Están precedidas por &B.

Las constantes numéricas pueden ser de simple o de doble precisión. Las de simple se representan en la máquina con una exactitud de seis dígitos, y las de doble, con catorce. A menos que se especifique otra cosa, una constante siempre será representada en doble precisión. No obstante, cualquier número en forma exponencial se tratará generalmente como simple precisión. Si necesita que un número de doble precisión se represente en forma exponencial, la letra "E" es reemplazada por la letra "D". Las constantes también se pueden poner en simple precisión posponiendo al número el signo "!".

De esta manera tenemos muchas formas de representar las constantes en BASIC MSX. Sin embargo, los números retienen el mismo valor a todo lo largo de la ejecución de un programa. Por eso sería útil contar con un medio que nos permita representar un número que pueda cambiar su valor según progrese el programa. Aquí es donde emplieza a ser útil el concepto de variable. Una VARIABLE es mejor imaginársela como una serie de bytes en memoria a los que el ordenador puede referirse por un nombre: el NOMBRE DE VARIABLE. Esta serie de bytes pueden representar una cadena o un número. El nombre de variable es utilizado de esta forma por el programador para acceder al número almacenado en la variable. Cuando damos un valor a la variable decimos que estamos ASIGNANDO un valor a la variable; para hacer esto podemos usar la sentencia LET de BASIC, o podemos usar el signo "=" aislado;

```
LET A=100
A=100
```

Ambas sentencias asignan un valor de 100 a la variable A. Cada tipo de variable requiere una cierta cantidad de espacio donde almacenar el número. La tabla siguiente lo muestra.

Tipo	**N.º de bytes**
Entero	2
Simple precisión	4
Doble precisión	8
Cadena	3 + 1 por carácter.

Nombres de variable

Un nombre de variables es un conjunto de caracteres alfanuméricos (es decir, compuestos por letras y/o números) de forma única para cada variable. Los nombres pueden incluir ocasionalmente "!", "#", "$" o "%" como último carácter del nombre, pero este carácter tiene un significado especial. Indica si la variable es entera, de cadena, de simple o de doble precisión. Estos caracteres se llaman de DECLARACION DE TIPO, y son los siguientes:

Carácter	**Tipo**
%	Entera
$	Cadena
!	Simple precisión
#	Doble precisión

Si no se da el carácter de definición de tipo en el nombre de variable, el ordenador la tomará generalmente como de doble precisión.

Creación de nombres

Bautizar las variables es bastante fácil; un nombre de variable puede ser de cualquier longitud, pero sólo son considerados por el ordenador los dos primeros caracteres, sin contar el carácter de declaración de tipo. Un nombre de variable siempre debe empezar por una letra; si lo hiciera por un número, el ordenador incorporaría el nombre de variable como una línea de programa. Así, A1 es legal, pero 1A no lo es. Para ver el efecto que tiene el que sólo los dos primeros caracteres sean significativos, teclee lo siguiente, terminando cada línea con RETURN:

```
NEW
TABLA=1
TASA=20
PRINT TABLA,TASA
```

Observará que aparece 20 y 20 en la pantalla, indicando así que el ordenador no puede diferenciar entre los dos nombres de variable. Tan pronto como el ordenador ha explorado los dos primeros caracteres de una variable y ha comprobado que ésta existe, no tiene en cuenta el resto del nombre. Obviamente esto puede causar problemas si no tenemos cuidado. Un detalle adicional es apuntar que los nombres de variable no pueden contener ningún nombre de función, comando o sentencia BASIC, y el ordenador no diferencia entre letras mayúsculas y minúsculas. Por ejemplo, *sprint* es un nombre erróneo de variable porque contiene la palabra *print*, que el ordenador lee como PRINT. Por razones similares, SINK y DATA son nombres erróneos de variable porque el primero contiene SIN y el segundo DATA. Un nombre de variable no debe empezar con las letras FN, ya que el ordenador creería que se está refiriendo a una función definida por el usuario. Por tanto, FNF es un nombre erróneo de variable, porque la máquina piensa que se está refiriendo a una función llamada F. No se preocupe por esa repentina introducción de la función definida por el usuario; se explicará muy pronto.

Respecto al carácter de declaración de tipo que hemos mencionado anteriormente, tenemos que z# y z1 son variables de doble precisión; z% es una variable entera, y z$ es una variable de cadena. Sin embargo, también podemos declarar el tipo de variable utilizando las sentencias DEF *var*. Hay cuatro: DEFINT, DEFDBL, DEFSTR y DEFSNG, seguidas cada una de ellas por una sola letra o por dos, separadas por "_".

Así, DEFINT a definirá todas las variables que empiecen con la letra a como variables enteras. De igual forma, DEFDBEL definirá las variables de doble precisión; DEFSNG, las de simple precisión, y DEFSTR, las de cadena. Un punto a destacar es que la utilización posterior de un carácter de declaración de tipo predominaría sobre las sentencias DEF *var* emitidas. Como ejemplo, demos una orden DEFINT a, y después asignemos la variable a# =1.234. Si imprimiésemos entonces a# veríamos que es una variable de doble precisión, como podríamos esperar por haber usado el signo "#". Sin embargo, una segunda variable, a1, sería todavía una variable entera. Pruebe el programa siguiente:

```
10 DEFINT A
20 A=1.2346
30 A#=1.234567
40 A!=1.2345
50 PRINT A,A!,A#
```

Ejecútelo y observe cómo se tratan de diferente manera las variables que tienen caracteres de declaración de tipo y las que no lo tienen.

La sentencia DEFINT A-Z ocasionará que todas las variables disponibles sean enteras a menos que se especifique otra cosa. Si desea que una variable sea de cadena, de doble o de simple precisión, debe especificarlo usando caracteres de declaración de tipo. De modo similar, DEFINT I-K definirá todas las variables que empiecen por las letras I, J o K como enteras. DEFSTR puede dar resultados imprevistos al imprudente; puede haber variables de cadena sin necesidad del signo "$". La sentencia A = "manzana" normalmente generaría un mensaje de error *type mismatch.* Sin embargo, después de una sentencia DEFSTR A, la instrucción A = "manzana" es correcta, pero la más usual, A = 1.234, no lo es. Nuevamente, el uso de caracteres de declaración de tipo predominará sobre la sentencia DEFSTR.

Variables de matriz

Una matriz es una estructura de datos disponible para el programador MSX. No obstante, si las estructuras de datos que hemos visto previamente son conjuntos de bytes, una matriz es un conjunto de números o cadenas a los que se puede acceder con el mismo nombre. Un término individual almacenado en una matriz se llama ELEMENTO, y todos los de una matriz contienen datos del mismo tipo. Cuando enciende su ordenador, la máquina le permite usar matrices de hasta diez elementos, numerados del 0 al 9, sin tener que informar al ordenador de su pretensión de usar matrices. Si desea obtener más elementos, entonces la matriz debe ser DIMENSIONADA mediante la sentencia BASIC DIM. Por tanto, DIM A(20) dimensionará una matriz llamada A para que tenga veintiún elementos, numerados del 0 al 20.

Cuando dimensionamos una matriz por primera vez, cada elemento tiene el valor 0 si aquélla es numérica, o cadena vacía si es de cadena. Asignar valores a un elemento de una matriz es muy fácil.

```
A(1)=1.234
A(2)=3
LET A(4)=23
```

A(0,4)	A(1,4)	A(2,4)	A(3,4)	A(4,4)
A(0,3)	A(1,3)	A(2,3)	A(3,3)	A(4,3)
A(0,2)	A(1,2)	A(2,2)	A(3,2)	A(4,2)
A(0,1)	A(1,1)	A(2,1)	A(3,1)	A(4,1)
A(0,0)	A(1,0)	A(2,0)	A(3,0)	A(4,0)

Figura 3.1.—*Representación de la matriz A: DIM A(5,5).*

Estas sentencias asignarán los valores apropiados a los distintos elementos de la matriz A. Las matrices están bajo la influencia de la sentencia DEF *var*, y también pueden tener caracteres de declaración de tipo. Una matriz puede ser unidimensional, como el ejemplo de la matriz A, o puede tener varias dimensiones, como B(10,6). La mejor manera de entender las matrices multidimensionales es pensar en ellas como un conjunto de cajas colocadas en una rejilla. Las diferentes cajas son los elementos de la matriz, y la figura 3.1 muestra un diagrama de parte de una matriz de este tipo. Puede intentar acceder a un elemento de una que no existe, tal como A(99), cuando sólo hemos dimensionado A para treinta elementos; entonces tendremos el error "índice fuera de rango" *(subscript out of range)*. El número usado para acceder a un elemento de una matriz se llama INDICE. El índice puede ser una constante, una variable o una expresión, y el error "índice fuera de rango" ocurre frecuentemente cuando una expresión da un valor demasiado alto. El máximo número de dimensiones que puede tener una matriz es 255, pero el número de elementos sólo está limitado realmente por la cantidad de RAM de su máquina.

Cambio de tipos

Si teclea:

```
A="Fede"
```

tendrá un error *Type Mismatch:* no puede, pues, poner una cadena en una variable numérica. Sin embargo, el BASIC MSX le permite convertir un tipo de número en otro. Si asigna a una variable de un tipo un valor contenido en una variable de otro diferente, obviamente ésta asumirá el valor, y ese valor se almacenará en la variable de acuerdo con su tipo. Por ejemplo:

```
A%=1.23456
PRINT A%
```

imprimirá 1. El número real ha sido así convertido a entero. Se ha truncado la parte fraccionaria y no se ha hecho ningún intento para redondear el número ni por defecto ni por exceso. Así, en doble precisión, cuando se asigna un valor a una variable de simple precisión, solamente se utilizarán seis dígitos. Durante la evaluación de expresiones, el grado de precisión aplicado al resultado es el más alto de los poseídos por las variables o las constantes de la expresión.

Expresiones en BASIC MSX

Cotidianamente la suma 2 + 3 es una expresión simple. En la adición, llevamos a cabo un proceso conocido como EVALUACION, y conseguimos un resultado o VALOR. En este caso, el resultado sería 5. En términos técnicos, describimos una

expresión como un conjunto de números constantes, variables y operadores, que pueden ser evaluados para conseguir un resultado. No es obligatorio que una expresión contenga todo lo dicho anteriormente.

1 + 2 + 3 + 4

es una expresión, como lo es

A + B + 1 + 2

Sin embargo, la mayoría de las expresiones contienen al menos un operador. Un OPERADOR puede alterar el valor de una variable o de una constante realizando alguna operación artimética o lógica sobre ella.

En la expresión

1 + 2

el operador es "+". Es un operador aritmético, y en la expresión

```
1+ ASC("A")
```

se dice que ASC( ) es un operador funcional. En total, hay cuatro familias principales de operadores disponibles para el programador MSX. Son:

- i Operadores aritméticos.
- ii Operadores relacionales.
- iii Operadores lógicos.
- iv Operadores funcionales.

Ahora lo más lógico es ver detalladamente cada familia por separado. Comencemos con aquellos que nos son más familiares: los operadores aritméticos.

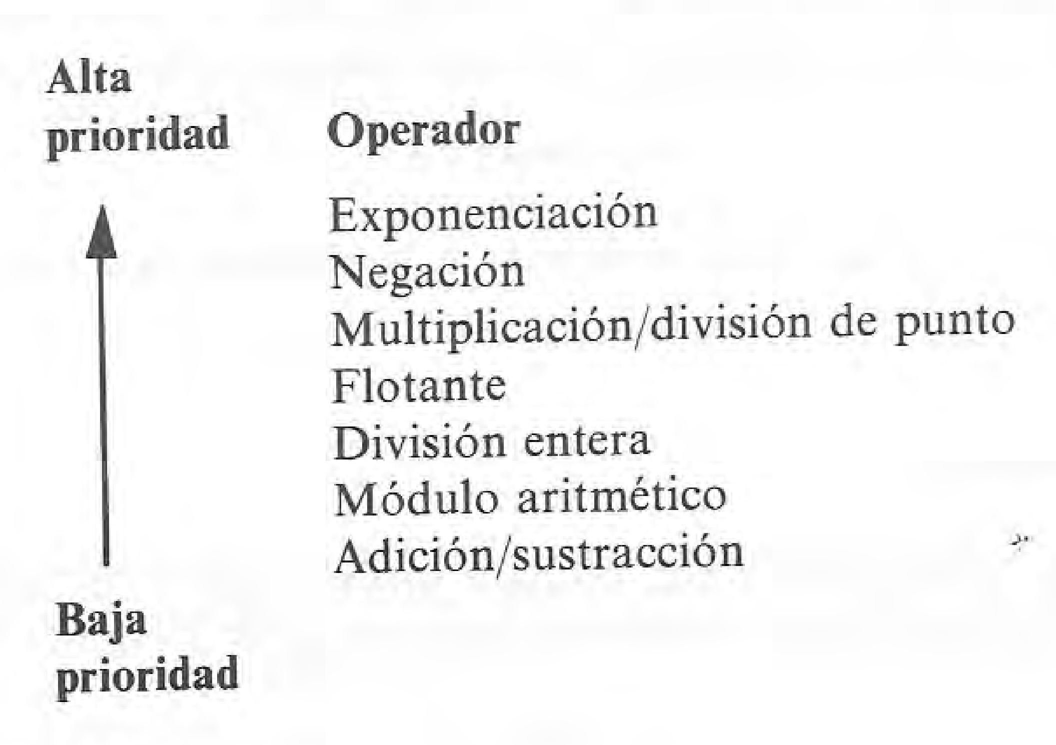
Operadores aritméticos

Hay ocho operadores aritméticos diferentes disponibles en el BASIC MSX. Son exponenciación (∧), negación (−), multiplicación (*), división (/), adición y sustracción, división entera (¥) y módulo aritmético. Veremos brevemente los operadores que son nuevos para nosotros. El ordenador evaluará las expresiones según unas reglas. Por ejemplo, la expresión

A + B*C

puede ser evaluada de dos maneras diferentes, como (A + B)*C o como A + (B*C). El ordenador evaluará la expresión de la segunda forma. Las expresiones pequeñas que van entre paréntesis serán evaluadas, sin embargo, de acuerdo con unas reglas

llamadas reglas de PRIORIDAD. El operador de multiplicación tiene mayor prioridad que el de adición. La figura siguiente nos muestra el orden de prioridad de los operadores:



Así, en la expresión

$$1+2\wedge 2$$

será evaluado primero el $2\wedge 2$, que dará 4, y entonces se añadirá 1, dando como resultado final 5. A pesar de ello, ¿qué ocurre si queremos evaluar la operación de suma antes que la de exponenciación? Pues usaremos paréntesis, como ya hemos visto. Para conseguir que la expresión se evalúe como queremos, escribiremos:

$$(1+2)\wedge 2$$

que dará como resultado 9. Las operaciones entre paréntesis se efectúan primero, pero todavía operan las reglas de prioridad dentro de ellos. Así, en las expresiones largas se suelen usar varios pares de paréntesis para indicar qué expresión se evalúa primero. A estos paréntesis, unos dentro de otros, se les llama PARENTESIS ANIDADOS. Un punto importante a recordar acerca de la anidación de paréntesis es que cada uno de apertura (( ) ha de estar acompañado por otro de cierre ( )). Si no se hace esto, el ordenador nos indicará el error. Pero, no obstante, no será el mensaje "¡falta un paréntesis!", sino *syntax error*. Metidos en el tema de errores, si usted coloca un operador en la expresión y no lo sigue de una constante, variable u otra expresión, se generará el error "falta de operando" *(missing operand)*. Un OPERANDO es algo sobre lo que trabaja un operador.

División entera

Con la división normal se suele obtener un número real. Sin embargo, la división entera trunca la parte fraccionaria del resultado. Así:

```
7¥2=3 y no 3.5
```

El símbolo "¥" indica división entera. Veamos aquí un par de puntos. Primero: antes de realizar la división, el ordenador convierte los operandos en enteros, entre −32768 y +32767; el resultado también se convierte en entero. Segundo: no intente que el ordenador divida por cero; es imposible y él lo sabe, ¡incluso si su programa piensa lo contrario! Debido a la conversión a enteros, la expresión

```
30.1¥01.4
```

será evaluada como 30 ¥ 0. El error debido a la división por cero también ocurre en la división real.

Módulo aritmético

En la escuela primaria enseñaban, no hace mucho, qué es el módulo aritmético. Pero no lo llamaremos así; para nosotros es el resto de la división.

```
10 MOD 4=2
```

El ordenador evalúa la expresión como 10 dividido entre 4 igual a 2 y de resto 2, y este resto es el valor que nos devuelve como resultado de la expresión. De forma similar,

```
10 MOD 5=0
```

Una vez en el tema de operadores aritméricos, merece la pena examinar un par de errores que se pueden dar. El primero es *Overflow*, donde el resultado de sus cálculos es demasiado grande como para que el ordenador pueda manejarlo. El segundo es *Type Mismatch*, donde ha intentado asignar un número a una variable de cadena. Usualmente se hace esto después de un comando DEFSTR, ya que se tiende a tomar por numéricas las variables que no tienen carácter de declaración de tipo.

Operadores relacionales

Se utilizan para comparar dos valores, expresiones, variables o constantes, y dan como resultado un 0 o un −1. En estos casos, a −1 se le considera VERDADERO *(true)* y a 0 FALSO *(falso)*. El resultado de una operación de este tipo puede usarse para controlar el flujo de programa, usando la sentencia BASIC IF que encontraremos pronto. Hay seis operadores relacionales en BASIC MSX, que son.

Símbolo	**Operador**
=	Igual
< >	Distinto
>	Mayor que
<	Menor que
> =	Mayor o igual que
< =	Menor o igual que

Veamos un par de ejemplos del uso de operadores relacionales. Un verdadero devuelve −1; un falso, 0.

```
PRINT (1=1)
```

retornará −1, porque es cierto que 1 = 1. La expresión

```
PRINT (1=2)
```

dará el valor 0 o falso. La expresión más compleja

```
PRINT (A-B)=1
```

solamente será verdadera si el valor de la expresión (A − B) es 1.

Se pueden asignar variables usando los operadores relacionales; obviamente, los valores asignados a estas variables serán 0 o −1.

Por tanto, la expresión

```
A=(1=2)
```

asignará a la variable A el valor 0. Cuando se combinan operadores relacionales y aritméticos, se evalúan primero las expresiones de tipo aritmético.

Operadores lógicos

Son ligeramente más complicados que los operadores anteriores, y pueden causar confusión. Por otra parte, cuando se utilizan correctamente son poderosas herramientas de programación. Por esta razón, ¡vamos allá! La manera más típica e interesante de utilizar operadores lógicos es para enlazar operadores relacionales. Los operadores lógicos disponibles en BASIC MSX son AND (y), OR (o), NOT (no), EQV (equivalencia), XOR (o exclusivo) e IMP. Los más usados comúnmente para unir expresiones relacionales son AND, OR y NOT, y por eso veremos primero estos tres y en ese orden. Cuando se usan de esta manera, la expresión que contienen los operadores lógicos y relacionales retorna un valor verdadero o falso. La expresión

```
PRINT (A=1) AND (B=2)
```

dará un valor verdadero sólo cuando ambas condiciones (A = 1 y B = 2) sean verdaderas. Un ejemplo más es

```
PRINT(A$="FEDE")AND(B$="PEPE")
```

Aquí, A$ debe contener "FEDE" y B$ "PEPE". La expresión

```
PRINT NOT(P)
```

tendrá un valor verdadero si P = 0, y falso en cualquier otro caso. Como 0 es el valor usado para representar "falso", podemos ver que NOT falso es verdadero.

Estos dos operadores, y OR, se emplean frecuentemente cuando nuestro programa tiene que tomar una decisión basada en el cumplimiento de varias condiciones. Las expresiones lógicas del tipo de las anteriormente mostradas pueden incluirse en sentencias IF para ayudar al control del flujo del programa.

El segundo uso de estas operaciones es examinar un byte según un modelo de bits. Se conoce como operación BINARIA, y es este aspecto del uso de operadores lógicos el que puede plantear problemas. En primer lugar, veamos cómo funcionan los operadores lógicos con un solo bit. El resultado de las operaciones sobre varias combinaciones de bits se muestra en las siguientes TABLAS DE VERDAD:

NOT

A	NOT A
0	1
1	0

AND

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

XOR

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

EQV

A	B	A EQV B
0	0	1
0	1	0
1	0	0
1	1	1

IMP

A	B	A IMP B
0	0	1
0	1	1
1	0	0
1	1	1

Expliquemos unas cuantas cosas antes de seguir adelante. XOR es contracción de la frase *exclusive OR* (OR exclusivo). Esta función sólo tiene el valor 1 si A OR B tienen valor 1, pero no si lo tienen los dos. También EQV es abreviatura de equivalencia; toma el valor 1 sólo si A = B. La función IMP se ha bautizado "importancia", ya que la expresión A IMP B toma un 1 si el valor de B es mayor o igual que el valor de A. ¿Cómo podemos aplicar esto a los números o variables? Bien. ¿Recuerda que los números se representaban como secuencias de bytes dentro del ordenador? Los enteros se almacenaban como números de 16 bits, ocupando 2 bytes. El bit situado más a la izquierda de la representación binaria de un entero MSX indica el signo, si es positivo o negativo. Por eso es por lo que los enteros MSX sólo pueden tomar valores entre −32768 y +32767. Deben ser enteros de este tipo todos los números a utilizar con las operaciones lógicas binarias. Así, cualquier expresión a utilizar en una operación binaria debe tomar un valor comprendido dentro de ese rango. Intentemos una adición binaria de dos números, 8 y 3.

```
PRINT 8AND3
```

devolverá el valor 0. Para ver de dónde procede esta situación, convirtamos 8 y 3 en sus equivalentes binarios y examinémoslos:

```
8 0000000000001000
3 0000000000000011
```

Apliquemos ahora la tabla de verdad a los números binarios; encontramos que no hay ningún 1 en una posición común a los dos números, y por eso apareció el resultado 0. Para evidenciarlo más, intente la operación con números binarios.

```
PRINT &B0001000 AND &B00000011
```

Ahora podemos comparar directamente las expresiones binarias. El comando binario OR puede ser empleado de una manera similar.

```
PRINT 8 OR 3
```

dará como resultado 11. Para comprenderlo mejor utilice la tabla de verdad de OR.

Cuando usamos números negativos en operaciones binarias hemos de tener cuidado debido al hecho de que los enteros negativos en BASIC MSX se representan en notación de complemento[2] a 2. Así, – 1 se representa en binario como 1111111111111111. Es decir,

```
PRINT -1 AND 2
```

devolverá como resultado 2. Uno de los cometidos de los operadores binarios es modificar los valores de ciertos **bits** dentro de bytes sin alterar el resto de los bits.

Operadores funcionales

Una función es un conjunto definido de operaciones que puede aplicarse a una expresión u operando. Hay dos tipos de funciones soportadas por el BASIC MSX. El primero y principal es la FUNCION INTRINSECA, funciones presentes en el ordenador desde el mismo momento en que la máquina es conectada. Incluyen funciones tales como SIN(*n*), ASC( ) y CHR$(*n*). El valor *n* puede ser aquí una expresión u operando, y se llama argumento de la función. Para la función ASC( ), el argumento ha de ser un carácter. Se dice que el argumento es "pasado" a la función, y que ésta devuelve un resultado. Si intenta pasar un argumento numérico a la función cuando ésta requiere uno de carácter o de cadena, aparecerá un error *Type Mismatch*. Este error será generado por la sentencia

```
A=SIN("Hola")
```

El argumento de una función puede ser otra función, siempre que se facilite a la última un argumento del tipo correcto. Así,

```
PRINT CHR$(ASC("A"))
```

imprimirá en la pantalla la letra "A".

La segunda clase de función soportada por el BASIC MSX es la "función definida por el usuario". Son funciones escritas por el programador para realizar una tarea específica. Veamos cómo podemos definir nuestras propias funciones.

Definición de funciones

Antes de que una función definida por el usuario pueda ser usada, se debe proveer al ordenador de una definición de la función, es decir, una lista de sentencias que representan la función. Una función definida por el usuario puede usar funciones intrínsecas en su definición, como veremos después.

La definición de una función se realiza la sentencia DEF FN. Es una abreviatura del inglés **Def***ine* **Fu***nction*, y la sentencia dice al intérprete BASIC que lo que sigue en la línea es una definición de función. Las definiciones de función deben ser siempre ejecutadas antes de llamar a la función. Lo siguiente es un ejemplo de una definición de función:

```
10 DEF FNtest 1 (a,b,c$)=SIN(a)+SIN(b)
```

Aquí, test es el nombre de la función, y ésta se limita a retornar como resultado la suma de los senos de los números a y b. a, b y c$ son los parámetros de la función, y juntos se les llama la lista de parámetros de la función. Cada parámetro en la lista se separa de los demás por medio de una coma. Estos parámetros son solamente operandos ordinarios; si son variables se les denomina según las reglas que rigen la denominación de variables. El nombre de función es llamado de igual manera que las variables. Es correcto tener una función con el mismo nombre de una variable, porque el ordenador trata estos nombres diferentemente.

La expresión situada a la derecha del "=" en la definición de la función puede ser solamente de la longitud de una línea de programa:

```
10 DEF FNtest 1 (a,b)=SIN(a)
20 +SIN(b)
```

generará un mensaje de error de sintaxis *(syntax error)* en la línea 20. Una función llamada test habrá sido definida, pero ésta sólo devolverá el seno de un número a. Los nombres de variable que aparecen en esta expresión sirven solamente para definir la función, y los cambios en sus valores no se reflejan en los de valores de las variables con el mismo nombre fuera de la definición. Si su función tiene una lista de parámetros, entonces las variables listadas en ella pueden ser incluidas como parte de la expresión, pero no es obligatorio.

Una función definida por el usuario es llamada con la sentencia FN. Recuerde que antes de que una función pueda ser llamada, la correspondiente sentencia DEF FN debe haber sido ejecutada. Para llamar a la función que hemos definido anteriormente teclearíamos algo semejante a

```
100 PRINT FNtest1(1,2,"Hola")
```

Es vital que el número de parámetros dados a la función sea el mismo que en la lista de parámetros que fue definida en la sentencia DEF FN. Asimismo, el tipo de los valores dados a la función cuando sea llamada deben ser los mismos que los tipos que fueron usados en la lista de parámetros cuando se definió la función. Si no se hace así, entonces se generará un error "tipo de variable incorrecta" *(type mismatch)*. Los parámetros dados de esta manera a la función son pasados de uno en uno sobre una base (el valor de las variables o constantes es pasado a la correspondiente variable en la lista de parámetros de la función llamada). Así, en el

ejemplo que acabamos de ver, a la variable a se le asignaría el valor 1; a b se le asignaría el valor 2, y c$ tomaría la cadena "Hola" asignada a ella. Si una variable es usada en la expresión, pero no está en la lista de parámetros, entonces en la evaluación de la expresión el valor usado es el valor corriente de la variable en el programa. La capacidad de pasar así parámetros a la función y de recoger los resultados retornados hacen de la función definida por el usuario una poderosa herramienta de programación. Si vamos a utilizar subrutinas (las veremos en el capítulo siguiente), entonces necesitaríamos fijar las variables usadas en la expresión en los valores apropiados, antes de que la subrutina sea llamada.

Por tanto, tendríamos un fragmento de programa como:

```
100 a=1
110 b=2
120 c$="Hola"
130 GOSUB 1000
```

Esto supone que el código en la línea 1000 es el mismo que pusimos en la definición de la función que usamos hace un momento. La llamada a la función equivalente es mucho más obvia:

```
100 resultado=FNtest(1,2,"Hola")
```

Si intenta llamar a la función antes de haberla definido usando la sentencia DEF FN, entonces se emite el mensaje de error "función del usuario indefinida" *(undefined user function)*. Las funciones aparecen en varios lugares de este libro; por tanto, podrá ver algunos ejemplos.

La sentencia DEF FN debe ser escrita dentro de un programa. Intentarlo en modo directo generará el mensaje de error "ilegal en modo directo" *(illegal direct)*, que indica que ha ejecutado un comando en modo directo que debe ser ejecutado únicamente desde dentro de una línea de programa.

Operaciones con cadenas

Aunque hemos mencionado variables de cadena, todavía no hemos discutido qué operaciones podemos hacer con ellas. Como no se puede hacer mucha aritmética con las cadenas de caracteres, el único operador aritmético que usamos con cadenas es "+".

Sin embargo, el operador no realiza la operación de adición, sino que une las cadenas. Este proceso es llamado CONCATENACION.

```
a$="ABC"
b$="DEF"
PRINT a$+b$
```

La cadena "ABCDEF" es impresa en la pantalla. Así, podemos formar una cadena larga de dos o más cadenas cortas.

```
C$=A$+B$
```

es correcto. Podemos, sin embargo, usar los operadores relacionales con cadenas.

Cuando el ordenador compara dos cadenas usando los operadores relacionales, lo hace carácter a carácter, comparando los códigos ASCII de cada uno de la cadena con el código de carácter correspondiente de la otra cadena. Si todos los códigos son iguales, entonces las cadenas son iguales. Sin embargo, puede ser encontrado un código en una cadena que es menor que el código correspondiente de la otra cadena; entonces la cadena con el código menor se dice que es menor que la otra cadena. Por ejemplo:

```
"AA"<"AB"
```

Cualquier espacio encabezando o intercalado es también examinado, y por tanto

```
"AA"<"AA"
```

La comparación es buena hasta que llegamos a la situación donde una cadena es más corta que la otra. ¿Qué ocurre ahora? Bien. La cadena más corta de las dos se dice que es menor que la más larga. Así,

```
"AB"<"ABC"
```

Como en el caso de los operadores relacionales aplicados a números, los valores verdadero y falso son retornados como resultado de una comparación relacional de dos cadenas. Los operadores lógicos como AND (y) pueden ser empleados para realizar funciones relacionales más complejas como

```
PRINT (A$="1" AND B$="2")
```

que solamente retornará un valor verdadero si A$ = "1" y B$ = "2". Podemos también usar la comparación de cadenas para controlar el flujo de programa del ordenador en conjunción con la estructura IF ... THEN ... ELSE.

Obviamente, las operaciones lógicas binarias son imposibles, pero una operación binaria puede ser realizada con el código ASCII de un carácter.

Ahora estamos preparados para echar una mirada al resto del BASIC MSX. No tenga miedo de experimentar programando su máquina. Esa es la mejor manera de aprender cómo conseguir los mejores resultados del ordenador. Si los ejemplos del texto son sugestivos, tecléelos; si encuentra una manera de hacer una determinada tarea diferente de la que hemos utilizado, experiméntela: ¡su método puede ser mejor!

NOTAS DEL CAPITULO

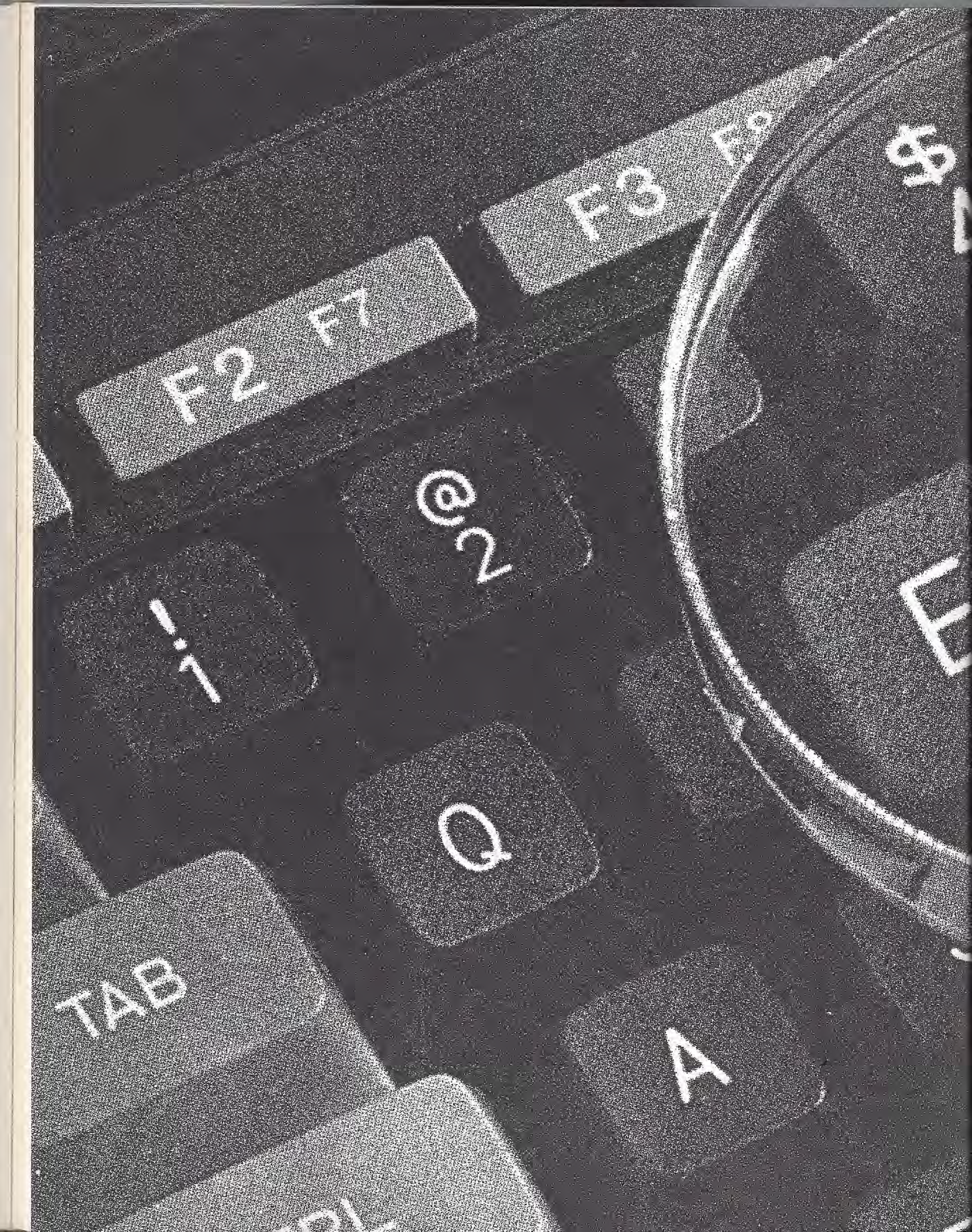
1.—Los códigos ASCII de los caracteres son los siguientes:

DECIMAL	HEXADECIMAL	CARACTER	SIGNIFICADO	CONTROL
0	00	NUL	nulo, ceros	
1	01	SDH	inicio, cabecera	A
2	02	STX	principio de texto	B
3	03	ETX	fin de texto	C
4	04	EDT	fin de transmisión	D
5	05	ENQ	petición	E
6	06	ACK	acuse de recepción	F
7	07	BEL	llamada	G
8	08	BS	retorno de una posición	H
10	0A	LF	descenso de una posición	J
11	0B	VT	tabulación vertical	K
12	0C	FF	página siguiente	L
13	0D	CR	retroceso de carro	M
14	0E	SD	código especial	N
15	0F	SI	código normal	O
16	10	DLE	escape transmisión	P
17	11	DC1	control del dispositivo periférico 1	Q
18	12	DC2	control periférico 2	R
19	13	DC3	control periférico 3	S
20	14	DC4	control periférico 4	T
21	15	NAK	acuse de recepción negativo	U
22	16	SYN	sincronización	V
23	17	ETB	fin de bloque de transmisión	W
24	18	CAN	anulación	X
25	19	EM	fin de soporte	Y
26	1A	SUB	sustitución	Z
27	1B	ESC	escape	
28	1C	FS	separador de archivo	
29	1D	GS	separador de grupo	
30	1E	RS	separador de registro	
31	1F	US	separador de elementos de información	
32	20	SP	espacio	
33	21	!		
34	22	”		
35	23	#	indicador numérico	
36	24	$		
37	25	%		
38	26	&		
39	27	’		
40	28	(		
41	29	)		
42	2A	*	(signo multiplicación)	
43	2B	+		
44	2C	,		
45	2D	–		
46	2E	.		
47	2F	/	(signo división)	
48	30	0		

DECIMAL	HEXADECIMAL	CARACTER	SIGNIFICADO	CONTROL
49	31	1		
50	32	2		
51	33	3		
52	34	4		
53	35	5		
54	36	6		
55	37	7		
56	38	8		
57	39	9		
58	3A	:		
59	3B	;		
60	3C	<		
61	3D	=		
62	3E	>		
63	3F	?		
64	40	@ (-at-)		
65	41	A		
66	42	B		
67	43	C		
68	44	D		
69	45	E		
70	46	F		
71	47	G		
72	48	H		
73	49	I		
74	4A	J		
75	4B	K		
76	4C	L		
77	4D	M		
78	4E	N		
79	4F	O		
80	50	P		
81	51	Q		
82	52	R		
83	53	S		
84	54	T		
85	55	U		
86	56	V		
87	57	W		
88	58	X		
89	59	Y		
90	5A	Z		
91	5B	[(o “Ä”)		
92	5C	/ (backslash) (u “Ö”)		
93	5D	] (o “Ü”)		
94	5E	(o “↑”)		
95	5F	— (o “→”)		
96	60			
97	61	a		
98	62	b		
99	63	c		
100	64	d		
101	65	e		
102	66	f		

DECIMAL	HEXADECIMAL	CARACTER	SIGNIFICADO	CONTROL
103	67	g		
104	68	h		
105	69	i		
106	6A	j		
107	6B	k		
108	6C	l		
109	6D	m		
110	6E	n		
111	6F	o		
112	70	p		
113	71	q		
114	72	r		
115	73	s		
116	74	t		
117	75	u		
118	76	v		
119	77	w		
120	78	x		
121	79	y		
122	7A	z		
123	7B	(o "ä")		
124	7C	(u "ö")		
125	7D	(o "ü")		
126	7E	(o " ")		
127	7F	DEL	borrado	

2.—La notación del complemento a dos se explica en el apéndice.


F2 F7
F3
@
2
!
1
Q
TAB
A



4 Almacenaje en cinta de cassette

La RAM de su ordenador MSX perderá todo registro de su programa tan pronto como la alimentación sea desconectada o teclee usted el comando NEW. Es obvio, por consiguiente, que necesitamos un método para hacer una copia permanente del programa que podamos usar para almacenarlo indefinidamente. Esto lo hacemos salvando una copia del programa en una cinta de cassette. Como somos capaces de salvar un programa BASIC en la cinta, también podemos salvar variables, tablas o bloques de bytes.

Cuando transferimos datos a la cinta desde el ordenador, son grabados en ella como una serie de tonos; cada tono representa un bit de un byte. Así, cada byte en la cinta es representado por ocho tonos (la frecuencia del tono dice al ordenador si el bit tiene un valor de 1 ó 0). Esta codificación de los datos en dos tonos diferentes de audio se llama codificación por cambio de frecuencia (en inglés, *frequency shift keying*) o, abreviado, FSK. Si pudiéramos "ver" los tonos de audio en la cinta, veríamos la figura 4.1.

Los tonos que se utilizan para representar unos o ceros dependen de la velocidad con que el ordenador salva los datos en la cinta. La velocidad de la transferencia de los datos se conoce como "velocidad de transmisión en BAUDIOS", y es una medida aproximada del número de bits enviados por segundo. Los ordenadores MSX pueden leer o escribir datos a 1.200 ó 2.400 baudios.

A 1.200 baudios, un "0" es representado en la cinta por un ciclo de una frecuencia de 1.200 Hz (1 Hz es un ciclo por segundo).

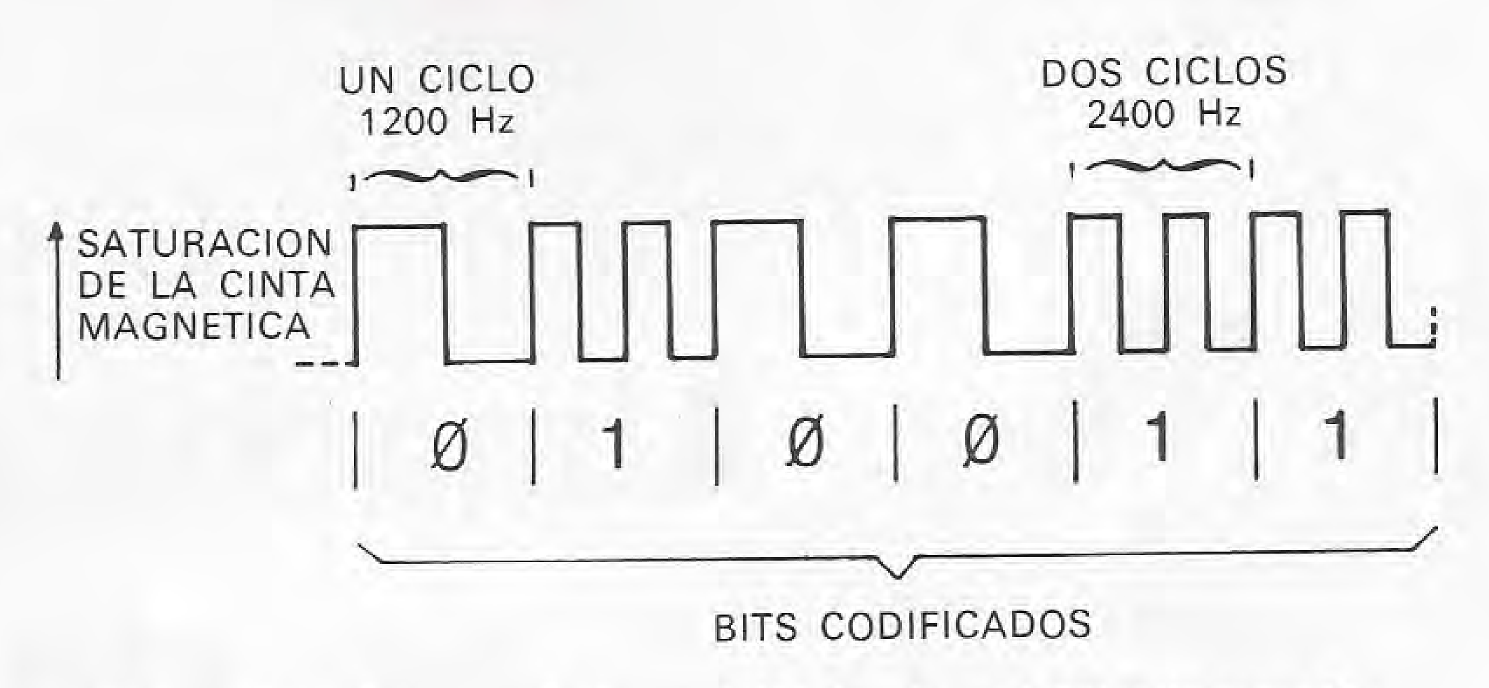


Figura 4.1.—*Datos grabados a 1.200 baudios (1.200 bits/segundo).*

A esta velocidad de transmisión, un "1" es representado por dos ciclos de 2.400 Hz. La velocidad de transmisión de datos adoptada generalmente por los ordenadores MSX es 1.200 baudios, y es segura con casi todos los cassettes.

La velocidad de 2.400 baudios facilitará la transferencia de datos a dos veces la velocidad de 1.200 baudios, pero no es muy fiable con algunos cassettes. Un "0" es ahora representado por un ciclo de una frecuencia de 2.400 Hz, y un "1" es representado por dos ciclos de una frecuencia de 4.800 Hz.

La velocidad de transmisión es seleccionable por el programador, y generalmente es de 1.200 baudios. La velocidad puede ser cambiada usando el comando CSAVE, como veremos pronto, o por la orden SCREEN, como veremos en un capítulo posterior. Leyendo datos del cassette, el ordenador puede decidir automáticamente qué velocidad de transferencia utilizar para leer la cinta.

El ordenador, tanto enviando datos al cassette como leyéndolos, puede controlar el motor del cassette, si éste tiene un conector para control remoto. De esta forma el ordenador puede activar el motor del cassette, enviar los datos a la cinta y, después, desactivarlo. Los comandos BASIC MOTOR ON y MOTOR OFF controlan el motor del cassette. MOTOR ON lo activará y MOTOR OFF lo apagará. Si tecleamos sólo MOTOR, el motor se apagará si estaba encendido, o se encenderá si estaba apagado. A esto se le llama TOGGLING. El motor es controlado por un relé del ordenador, y cuando usted da una instrucción MOTOR, o usa comandos que controlan el cassette, puede oírse el "click" del relé, sonido que emite siempre que se enciende o apaga el motor.

Salvar programas

Lo primero para lo que la mayoría de la gente desea usar un cassette con un ordenador es para salvar sus programas; en los ordenadores MSX hay dos maneras de almacenar los programas BASIC en la cinta. Por tanto, antes de que vayamos más lejos examinemos estos dos métodos. Un bloque de información que es salvado en la cinta de cassette proveniente del ordenador se llama FICHERO (en inglés, *file*). Un programa puede ser almacenado en cinta como un fichero de caracteres ASCII, en cuyo caso el comando BASIC PRINT se salvará como P, R, I, N, T;

esto es, como letras individuales, o como un fichero INDEXADO (en inglés, *tokenized*), en cuyo caso la instrucción BASIC PRINT es almacenada como un solo número. Este número se dice que es el INDICE *(token)* del comando, y cada función o sentencia BASIC tiene un índice. Para tener una mejor idea acerca de los índices, veamos cómo se almacenan las líneas de programa BASIC por el sistema MSX:

```
10 REM programa de prueba
```

es almacenado como una serie de números en la RAM del ordenador. Estos números son:

```
20,192,10,0,143,32,116...
```

El 20 y el 192 forman un puntero para el principio de la siguiente línea del programa BASIC. Este puntero es almacenado como un número entero de 2 bytes, con el byte de menos valor[1] almacenado primero. Así, en este ejemplo particular, la siguiente línea de programa es almacenada en la dirección 20+(192*256), o la dirección 49172. En esta dirección encontraremos otro puntero de 2 bytes que señala al principio de la siguiente línea. El 10 y el 0 son el número de línea, otra vez almacenado el byte de menor valor primero.

Finalmente, llegamos al texto de la línea BASIC. Observe cómo la sentencia REM es representada por el índice 143 y en el texto del comentario no se muestran todos los códigos ASCII, ya que no se considera demasiado importante en este momento. Esta es la forma en que se salva el programa si usamos el comando CSAVE. La sentencia

```
CSAVE"test"
```

salvará el programa que reside actualmente en el ordenador como un fichero (indexado) llamado "test". Una razón obvia para salvar programas de este modo es que tarda menos en transferirse a la cinta (solamente se transfiere 1 byte para la sentencia PRINT en lugar de los 5 bytes que habrían de ser transferidos si el programa fuera salvado como un fichero ASCII). La velocidad de transferencia usada para el comando CSAVE es la tomada por defecto, 1.200 baudios, o la velocidad actual, seleccionada por el último comando SCREEN. En cualquier caso, si desea especificar la velocidad de transferencia ha de añadir un parámetro extra a la orden CSAVE. Por ejemplo:

```
CSAVE"test",1
```

salvará el programa a 1.200 baudios, y el comando

```
CSAVE"test",2
```

salvará el programa a 2.400 baudios. La velocidad de transferencia fijada en una orden CSAVE es solamente aplicable a ese comando y no tiene efecto sobre la ve-

locidad de transferencia de operaciones sucesivas de transferencia de datos. Observe que después de pulsar la tecla RETURN (o ENTER), tras haber tecleado una instrucción CSAVE, los datos empiezan a transferirse a la cinta inmediatamente. Por eso es necesario pulsar las teclas RECORD y PLAY de su cassette antes de pulsar RETURN. El primer fragmento de datos que se escribe en la cinta es un simple tono llamado director (LEADER). "Escuchándolo", el ordenador puede determinar la velocidad de transferencia con que fueron grabados los datos cuando queramos cargar de nuevo el programa. Seguidamente viene la cabecera (HEADER), que contiene información acerca del fichero que va a continuación, y finalmente los datos que representan el programa escrito en la cinta.

Después de presionar PLAY y RECORD en el cassette, el motor solamente se activará si la clavija que une el ordenador con el control remoto del cassette no está conectada. Pero si lo está, entonces el motor no empezará a funcionar hasta que la tecla RETURN sea pulsada tras haber tecleado un comando CSAVE. Después de que el programa ha sido salvado en la cinta, el cursor volverá a la pantalla y el motor de la cinta, si estaba controlado por el ordenador, se parará.

Veamos ahora cómo se salvan ficheros de caracteres ASCII para representar programas BASIC. Como cada carácter es salvado en la cinta, los programas salvados como ficheros ASCII tardan más en grabarse y además ocupan más espacio. Sin embargo, salvar programas como ficheros ASCII tiene sus ventajas. Los programas salvados de esta manera pueden ser usados como datos para que trabajen sobre ellos otros programas. Enseguida veremos con más detalle los ficheros de datos. Pero la principal ventaja de salvar un programa como un fichero ASCII es que dicho programa puede ser añadido a otro que está ya en memoria.

Este proceso, conocido como MERGING (FUSION), de los dos programas es bastante útil. Nos permite almacenar en la cinta secciones de programas usadas comúnmente e incorporarlas dentro de otros programas con el mínimo esfuerzo. Para salvar un programa en un formato ASCII en BASIC MSX usamos el comando SAVE.

```
SAVE"test"
```

salvará el programa en formato ASCII en la cinta en un fichero llamado "test". La instrucción

```
SAVE"CAS:test"
```

también salvará el programa en el cassette en formato ASCII. La parte del nombre de fichero CAS: se llama DESCRIPTOR DE PERIFERICO *(device descriptor)* y permite que los programas sean transferidos a otros periféricos en formato ASCII. Por ejemplo, el comando

```
SAVE"CRT:test"
```

realizará una operación que es similar a la operación LIST; el programa será mostrado en la pantalla de televisión. Si tenemos un descriptor de periférico "LPT:" en el nombre de fichero, entonces el fichero será listado en una impresora, si estu-

viera conectada, igual que si hubiéramos emitido el comando LLIST. La velocidad de transferencia usada en estas transferencias de datos es la fijada por el último comando SCREEN o 1.200 baudios. Seleccionar una velocidad de transferencia con el comando SAVE es difícil, pero no con la instrucción CSAVE.

El nombre de fichero usado puede ser una variable de cadena; así los comandos

```
A$="CRT:test"
SAVE A$
```

listarán el programa actual en la pantalla. Si repitiésemos la operación anterior con A$ igual a "CAS:test", entonces el programa sería escrito en la cinta como un fichero ASCII llamado "test". El nombre de fichero usado en las sentencias CSAVE puede ser también una variable de cadena. Los comandos

```
A$="test"
CSAVE A$
```

salvarán el programa en la cinta en forma indexada con el nombre de fichero "test".

Finalmente, destacaremos una característica de la orden SAVE. Cuando escribimos el fichero, el último código ASCII que se envía a la cinta es el caracter 26. Se envía para que el ordenador reconozca el final del fichero ASCII cuando, al volverlo a cargar, lo encuentre. Por esta razón el CHR$ 26 se llama marca de fin de fichero *(end of file)*.

Carga de programas

Para cargar un programa hemos de utilizar dos comandos diferentes, dependiendo del método empleado para salvarlo. Si el programa se salvó con la orden CSAVE, cuando volvamos a cargarlo se usará la sentencia CLOAD. El comando

```
CLOAD
```

por sí mismo cargará el siguiente programa BASIC indexado que se encuentre en la cinta. De todas formas, podemos haber salvado varios programas en la misma cinta, y por eso se suele usar CLOAD con un nombre de fichero, como sigue:

```
CLOAD"test"
```

cargará el programa "test" cuando lo encuentre en la cinta. "Test" debe ser, desde luego, un fichero indexado salvado utilizando CSAVE. Los ficheros encontrados mientras el ordenador está buscando "test" serán ignorados[3]. En ambos casos, tan pronto como se ha encontrado *(found)* el fichero correcto, el mensaje

```
Found: test
```

es visualizado en la pantalla. A continuación se procederá a cargarlo. Tan pronto como empiece la carga, el programa que previamente estaba en el ordenador se pierde. En cuanto el programa ha sido cargado, el ordenador vuelve al modo directo. Por esta razón, CLOAD no es muy utilizado como una sentencia en un programa, ya que la ejecución del programa finalizaría. CSAVE puede ser usado en una línea de programa, y tras haber sido salvado continúa la ejecución del mismo.

Si el programa ha sido salvado en formato ASCII con el comando SAVE, entonces debemos cargarlo de nuevo usando la orden LOAD. Este es el equivalente del comando CLOAD, pero trabaja en ficheros que han sido salvados como ficheros ASCII. Así,

```
LOAD"CAS:test"
```

buscará un fichero con formato ASCII llamado "test". Pero si desea que simplemente sea cargado el siguiente fichero que encuentre con formato ASCII, entonces la instrucción oportuna es

```
LOAD"CAS:"
```

La sentencia

```
LOAD"CAS:test",R
```

cargará el fichero ASCII llamado "test" y ejecutará un comando RUN, por lo que empezará a rodar el programa. La "R" es el único parámetro que usted puede dar a una orden LOAD.

Verificación

Sería muy bonito pensar que nuestros ordenadores son infalibles. Desgraciadamente, no es así. Salvar un programa en la cinta es muy problemático; por ejemplo, el cassette puede necesitar una limpieza, puede ser ruidoso o funcionar a una velocidad incorrecta, o cualquiera de una docena de problemas que podrían aparecer durante la grabación. Por tanto, antes de dar un comando NEW a la máquina sería útil conocer si el ordenador ha grabado una copia fiable del programa en la cinta, que sea cargable en una futura ocasión. La sentencia que se usa para hacerlo es CLOAD?, y opera comparando los bytes leídos de la cinta con los que están en la memoria del ordenador. Este proceso se llama VERIFICACION; verdaderamente en algunos ordenadores el comando CLOAD? es llamado VERIFY *(verificar)*. La sintaxis de la instrucción es:

```
CLOAD?"test"
```

Esta orden buscará el fichero "test" en la cinta y lo comparará con el programa que actualmente está en memoria.

Fusión

El último comando que utilizamos para manipular ficheros en cinta representando programas se llama MERGE. Cuando progrese en la programación, acumulará pronto una colección de subrutinas que son útiles en la mayoría de los programas. Por eso es también útil haberlas salvado en la cinta de tal manera que puedan ser incorporadas en cualquier programa. Para salvarlo utilizamos la orden SAVE, y cuando deseemos incorporarlas en un programa que está ya parcialmente escrito usamos la sentencia MERGE. La instrucción

```
MERGE"CAS:test"
```

buscará en la cinta hasta que un fichero llamado "test" sea encontrado. Este fichero sería un programa o una sección de programa salvada en formato ASCII. Cuando el fichero ha sido leído, las líneas de programa que estaban representadas en él habrán sido añadidas al programa. Cualquier línea que estuviera en el programa residente en el ordenador en el momento de la fusión, con números de línea idénticos a las líneas del fichero, serían reemplazadas por las líneas leídas del fichero. Así que ¡tenga cuidado con los números de línea en los ficheros ASCII! Cualquier sección del programa que haya de salvarse por este procedimiento debe ser renumerada para darle números de línea claramente altos. Ello reducirá las posibilidades de conflicto con los números de línea. Tan pronto como la operación de fusión termina, el ordenador regresa al modo directo y espera más comandos.

Si se omite el nombre de fichero, como en

```
MERGE"CAS:"
```

entonces el primer fichero que se encuentre será fundido con el programa de memoria. Recuerde que, para que un fichero esté disponible para la fusión, debe haber sido salvado con el comando SAVE.

Así como almacena programas en la cinta, el BASIC MSX nos permite almacenar variables y bloques de bytes. Estudiaremos cada caso por orden, empezando con la grabación de variables en la cinta.

Ficheros de datos

Como ya hemos visto, el BASIC MSX nos provee de una amplia variedad de estructuras de datos en las que podemos almacenar números o cadenas (alfanuméricas). Sin embargo, cuando quitamos la corriente, o ejecutamos una instrucción CLEAR o RUN, se pierden los valores de las variables. Pero usando el sistema de cinta de los micros MSX para escribir estos datos en ella podemos tener una grabación permanente de las variables utilizadas en un punto concreto del programa. Los valores de las variables se escriben en la cinta, con un nombre de fichero, y pueden ser cargados de nuevo en cualquier momento. A un fichero que tenga variables numéricas o de cadena se le llama "fichero de datos". Veamos una aplicación específica en la

que es esencial ser capaces de salvar datos usados en el programa. Imagine que estamos escribiendo un programa que almacene nombres, direcciones y números de teléfono. Podemos tener tres tablas de cadena, nom$(50), dir$(50) y tlf$(50), que tienen nombre, dirección y teléfono, respectivamente, de una determinada persona. Cada nombre y su correspondiente dirección y teléfono se conocen en conjunto como un registro (RECORD). El principio de este fichero cuando sea escrito en la cinta podría verse como la figura 4.2.

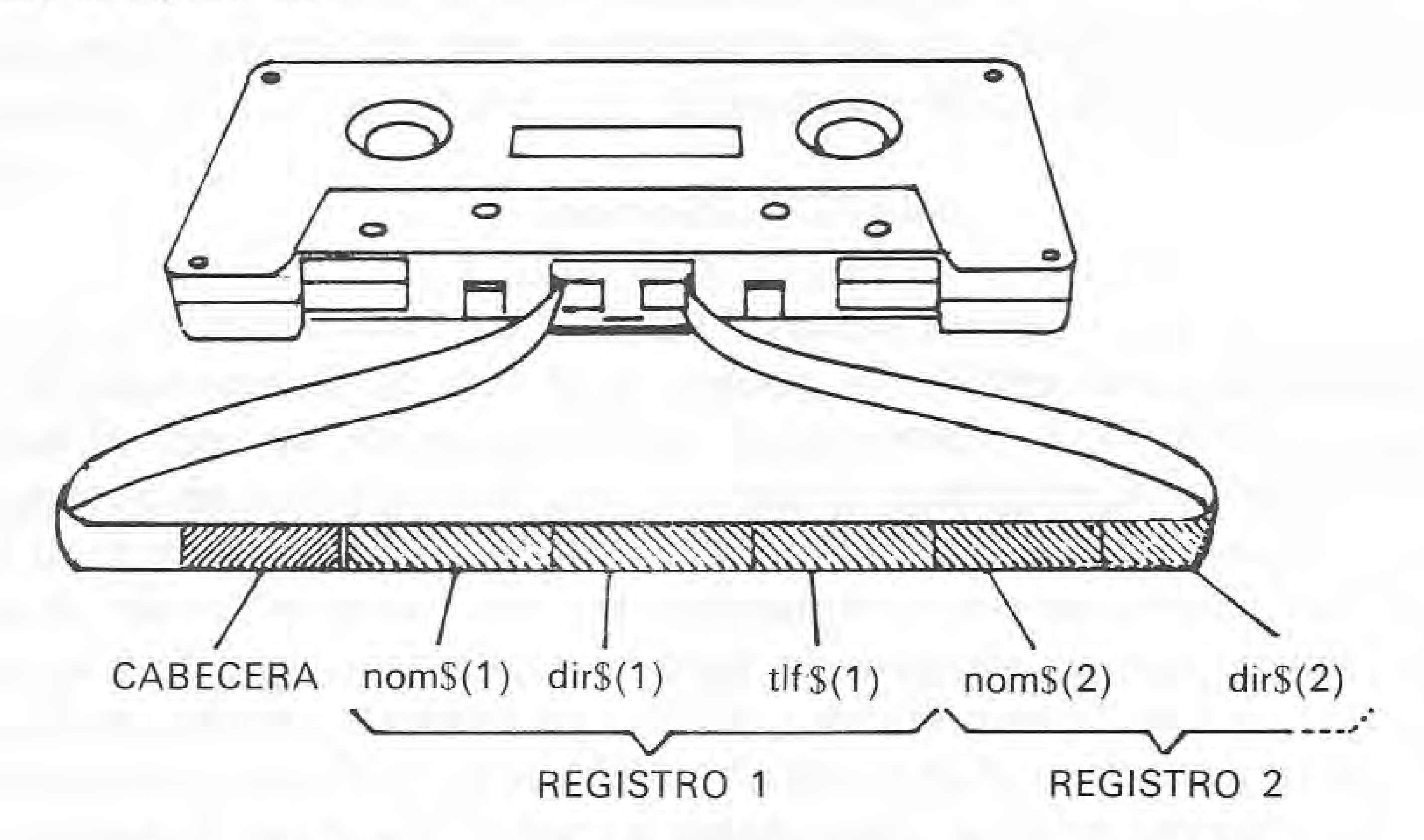


Figura 4.2.—*Esquema de ficheros en cinta (ficheros secuenciales).*

Por tanto, ¿cómo podemos conseguir datos contenidos en una variable en la cinta? Examinemos las sentencias BASIC que podemos usar para escribir datos en la cinta.

Apertura de ficheros

Antes de que podamos escribir datos en la cinta debemos abrir (OPEN) un fichero. Esta operación obliga al ordenador a preparar un área de memoria, para actuar como un tampón (BUFFER)[4], en donde los datos son escritos antes de ser enviados a la cinta. También se crea un BLOQUE DE CONTROL DE FICHERO *(file control block)*. Es un área de memoria que el ordenador usa para asegurarse de que la operación con el fichero sigue adelante tan fácilmente como sea posible. Los datos del *buffer* sólo son escritos en la cinta cuando está lleno, o cuando señalamos al ordenador que el fichero ha terminado. También es necesario abrir un fichero antes de que podamos leer datos en él.

La sentencia que usamos para abrir un fichero es de la forma

```
OPEN "CAS:nombre de fichero" FOR operacion AS #numero
```

El nombre del fichero es aquel por el que será salvado en la cinta. Observe cómo tenemos de nuevo la oportunidad de usar descriptores de periféricos para especificar dónde han de ser escritos los datos. Una vez hemos abierto un fichero, po-

demos leer o escribir datos en él. Así, la operación de la sentencia anterior puede ser entrada *(input)* o salida *(output)*. Si abrimos un fichero para salida, los datos son transferidos del ordenador a la cinta. Si se abre para entrada, la transferencia se dará en sentido inverso, es decir, los datos irán desde la cinta al ordenador. El número de las sentencias anteriores sirve como identificador específico del fichero mientras está abierto, y se usa siempre que necesitemos acceso al fichero. Así, en la instrucción

```
OPEN "CAS:test" FOR OUTPUT AS #1
```

abrirá el fichero de cinta llamado "test" con el número de fichero 1. El número de fichero debe estar entre 1 y 15, y por tanto es posible tener abierto más de un fichero a la vez. Si queremos fijar un tope de ficheros abiertos al mismo tiempo, usaremos un comando llamado MAXFILES. La sentencia

```
MAXFILES=6
```

le permitirá tener solamente seis ficheros abiertos a la vez, usando como números de ficheros del 1 al 6.

```
MAXFILES=0
```

no permitirá tener abierto NINGUN fichero. Después de que sea ejecutada una sentencia de este tipo, las únicas operaciones de cinta disponibles son las de salvar y cargar programas.

En la sentencia OPEN, "CAS:" puede ser reemplazado por "CRT:" o por "LPT:". De nuevo, para que "LPT:" funcione correctamente debe haber una impresora conectada al ordenador. La sentencia

```
OPEN"CRT:test" FOR OUTPUT AS #1
```

causará que todo lo que escribamos en el fichero número 1 aparezca en la pantalla.

Escritura de datos en los ficheros

Las sentencias empleadas aquí son similares a aquellas que usamos para imprimir datos en la pantalla. En lugar de PRINT tenemos PRINT #, y en lugar de PRINT USING tenemos PRINT #, USING. En cada caso al carácter "#" le seguirá un número de fichero. Así, la sentencia

```
PRINT #1,35
```

imprimirá el valor 35 en el fichero número 1. La instrucción

```
PRINT #1, USING "#.##";A
```

imprimirá el valor de la variable A en el fichero con dos dígitos después del punto decimal. Recuerde que antes de ejecutar cualquiera de estos comandos el fichero que va a recibir los datos debe estar abierto para salida. De manera similar, se puede acceder a los ficheros abiertos para entrada usando INPUT #, LINE INPUT # e INPUT$. Así, la sentencia

```
INPUT#1,A
```

leerá un valor numérico del fichero número 1 y asignará el valor leído a la variable A. La instrucción

```
A$=INPUT$(3,#1)
```

originará que el programa espere hasta que se hayan leído tres caracteres del fichero número 1. Si desea más detalles, vea la parte correspondiente al comando INPUT$.

INPUT #número de fichero, variable, trabajará de manera similar a la sentencia INPUT ya estudiada, pero no generará ningún *prompt*. Si lleva a cabo una entrada para asignar una cadena a una variable, se aplican las mismas consideraciones que en la sentencia INPUT normal. El comando LINE INPUT # leerá todos los caracteres que encuentre en el fichero hasta encontrar el carácter 13. Entonces se lee este carácter final y se asigna la cadena resultante a la variable de la instrucción LINE INPUT #. Enseguida veremos algunas aplicaciones donde se usan estas sentencias.

Cierre de ficheros

Al terminar de usar un fichero, independientemente de si estaba abierto para entrada o para salida, tenemos que CERRARLO. Esta operación es vital en los ficheros utilizados para salida de datos, ya que, si hay datos en el *buffer*, al cerrar el fichero nos aseguramos de que se escriben estos datos en la cinta. El acto de cierre de un fichero también libera la RAM utilizada por el ordenador para el *buffer* y el bloque control de fichero, de tal manera que puede ser usada para otros propósitos. Una vez cerrado el fichero, hay que reabrirlo (volver a abrirlo) antes de poder acceder a él de nuevo. La instrucción

```
CLOSE #1
```

cerrará el fichero que tiene asociado el número 1. Si se omite el "#1" se cerrarán todos los ficheros abiertos en ese momento.

Ejemplos del uso de ficheros de datos

En este apartado veremos cómo pueden usarse en programas BASIC los comandos y sentencias estudiados últimamente. El primer programa es un ejemplo trivial, pero que ilustra los puntos principales.

```
10 REM Escritura de un fichero en cinta
20 OPEN "CAS:test" FOR OUTPUT AS #1
30 FOR I=1 TO 10
40 PRINT #1,I
50 NEXT I
60 CLOSE #1
```

La línea 20 abre el fichero en cinta para salida. Las 30 y 50 configuran un bucle de tipo FOR-NEXT, y en la 40 se escriben los datos en la cinta. La línea 60 cierra el fichero para finalizar la operación. Enseguida veremos cómo podemos reemplazar el descriptor de periférico y el nombre de fichero por una variable de cadena en los comandos utilizados para grabar programas. Pues podemos hacer lo mismo aquí. Si insertásemos una línea 15, con A$="CAS:test", podríamos cambiar la línea 20 por OPEN A$ FOR OUTPUT A$ #1, y sería perfectamente correcto.

Es el momento adecuado para observar la representación de los datos en los ficheros de cinta. Si cambiamos el descriptor de periférico por "CRT:", veremos los datos escritos en la pantalla. Como comprobará, los números se escriben en la pantalla con el formato usado por el comando de impresión estándar. Así, en el ejemplo que acabamos de ver, los dígitos se escriben en la cinta separados por RETURN. Las cadenas también serán escritas en la cinta igual que se escriben en la pantalla.

El fichero que acabamos de escribir en la cinta puede ahora ser leído y sus datos transferidos al ordenador. El programa siguiente es una demostración de lo expuesto.

```
10 REM Lectura de un fichero
20 OPEN "CAS:test" FOR INPUT AS #1
30 FOR I=1 TO 10
40 INPUT #1,J
50 PRINT J
60 NEXT I
70 CLOSE #1
```

Tan pronto como sea localizado el fichero en la cinta, el bucle FOR-NEXT leerá 10 números del fichero usando la instrucción INPUT #1,J de la línea 40. El valor leído es impreso en la pantalla. Un rasgo interesante de cómo el BASIC MSX lee datos de los ficheros en cinta es que la línea 40 puede reescribirse como

```
40 INPUT #1,J$
```

o como

```
40 LINE INPUT #1,J$
```

Si se hace esto, y se altera la línea 50 escribiendo PRINT J$, observaremos que J$ contiene una representación alfanumérica del número que el ordenador acaba de leer de la cinta. Hay una explicación simple: los datos del fichero aparecen igual que si hubieran sido introducidos en respuesta a una sentencia *input* normal. Con los valores numéricos, el primer carácter distinto de un espacio, de un *return* o de un avance de línea que la sentencia INPUT encuentra en un fichero es tratado como el principio de un número. Entonces, cada carácter siguiente es tratado como parte del número hasta que se vuelva a encontrar un *return,* un avance de línea, un espacio o una coma. En este caso, cuando el ordenador espera la entrada de una cadena, el primer carácter que no es un espacio, un *return* o un avance de línea, se trata como el principio de la cadena. Si este primer carácter leído es "''" (unas comillas), la cadena leída comprenderá todos los caracteres hasta las siguientes comillas. Si el primer carácter leído no son unas comillas (''), la cadena leída constará de todos los caracteres hasta el siguiente *return,* avance de línea, coma o 255 caracteres. Si deseamos leer cadenas que contienen comas, usaremos la sentencia LINE INPUT #. Con ella, la cadena leída constará de todos los caracteres hasta el siguiente *return* y avance de línea, inclusive. Este comando es muy útil cuando se hayan escrito datos en cinta que contienen comas, y también puede ser usado para leer una línea de un programa BASIC que haya sido salvado como un fichero ASCII, y asignarla a una variable de cadena.

Si construimos un fichero repleto de variables de cadena, usando el programa listado a continuación, podemos ver qué ocurre cuando intentamos leer un dato alfanumérico usando una sentencia *input* que espera un valor numérico.

```
10 OPEN "CAS:test" FOR OUTPUT AS #1
20 FOR I=1 TO 10
30 PRINT #1,"Hola"
40 NEXT I
50 CLOSE #1
```

El fichero de cadenas producido por el programa anterior puede leerse usando INPUT #1,A$. Por ejemplo:

```
10 OPEN "CAS:test" FOR INPUT AS #1
20 FOR I=1 TO 10
30 INPUT #1,A$
40 PRINT A$
50 NEXT I
60 CLOSE #1
```

Sin embargo, reemplazando la línea 30 por INPUT #1,A y cambiando la línea 40 por PRINT A nos dará resultados totalmente diferentes. No se genera ningún error, pero se asigna el valor 0 a la variable. Incluso si la cadena es algo así como "1.234", el valor numérico sigue siendo 0.

Evidentemente los datos de los ficheros deben ser leídos de la misma manera en que fueron escritos. Por otra parte, los ficheros de datos pueden, desde luego, contener una mezcla de ambos tipos de datos, como nos demuestra el siguiente programa.

```
10 INPUT "¿Cuantas cadenas?";N
20 OPEN "CAS:cadena" FOR OUTPUT AS #1
30 PRINT #1,N
40 FOR I=1 TO N
50 INPUT A$
60 PRINT #1,A$
70 NEXT I
80 CLOSE #1
```

La línea 10 pide el número de cadenas que deseamos escribir en la cinta. En la línea 20 se abre el fichero y lo primero que hacemos es escribir el número de cadenas que va a contener. Un bucle FOR-NEXT se encarga de aceptar estas cadenas por teclado e imprimirlas en el fichero. Finalmente, se cierra el fichero. Con lo cual tenemos un fichero que contiene datos numéricos y de cadena. El fichero producido por esta rutina puede leerse con el programa siguiente:

```
10 OPEN "CAS:cadena" FOR INPUT AS #1
20 INPUT #1,N
30 FOR I=1 TO N
40 INPUT #1,B$
50 PRINT B$
60 NEXT I
70 CLOSE #1
```

Observará que hemos leído y asignado la cadena a B$; la variable que contiene los datos leídos de la cinta no tiene por qué tener el mismo nombre que la que se utilizó para escribir el fichero en la cinta cuando éste fue creado.

Uso de variables en sentencias OPEN

El nombre de fichero y el descriptor de periférico pueden ser reemplazados, si se desea, por una variable de cadena. Anteriormente hemos visto en este mismo capítulo cómo hacerlo. El número de fichero puede ser reemplazado también por una variable numérica. La instrucción

```
OPEN "CAS:test" FOR OUTPUT AS A
```

es correcta siempre que el valor de A esté entre 1 y el valor de MAXFILES. Cuando se abre un fichero de esta manera, las sentencias INPUT # o PRINT # también pueden indicar el número de fichero con una variable. Aquí, el nombre de variable que se usó en la sentencia OPEN simplemente reemplaza al dígito. Por tanto,

```
PRINT #A,"Hola"
```

imprimirá la cadena "Hola" en el fichero. Si se usa la variable de esta manera, ¡debemos tener cuidado de que el valor de la variable que se usa como número de fichero no cambie mientras éste esté abierto!

Como ejemplo final del uso de ficheros de datos en cinta se lista a continuación una subrutina, que escribiría en la cinta los datos de nuestros imaginarios nombre, dirección y número de teléfono:

```
1000 REM Subrutina para salvar un fich.
1010 OPEN "CAS:datos" FOR OUTPUT AS #1
1020 FOR I=1 TO 50
1030 PRINT #1,nombre$(I)
1040 PRINT #1,direccion$(I)
1050 PRINT #1,telefono$(I)
1060 NEXT I
1070 CLOSE #1
1080 RETURN
```

Aquí usamos simplemente un bucle FOR-NEXT para escribir en la cinta los elementos de las matrices. Cuando volvamos a leerlos, es necesario que las matrices que van a contener los datos hayan sido dimensionadas correctamente, con espacio suficiente para acomodar todos los datos que hubiera en el fichero. En el ejemplo anterior, si leemos los datos y se asignan a matrices que han sido dimensionadas sólo para veinte elementos, tan pronto como intentemos asignar el elemento 21 de la cinta tendríamos un error *subscript out of range.*

Ahora posee información suficiente para escribir programas simples de manejo de ficheros. Sin embargo, habrá observado que no puede leer, por ejemplo, el registro 21 sin haber leído antes el 20, y éste, a su vez, sin haber hecho lo mismo con los anteriores. A los ficheros escritos de esta manera se les llama ficheros secuenciales, ya que el acceso a los registros se hace de manera secuencial, uno detrás de otros, según la posición que ocupan en la cinta.

Salvar bytes

Una vez empiece a escribir programas en código máquina querrá salvarlos en la cinta. Todo cuanto hemos escrito en ficheros de cinta hasta ahora ha estado estructurado de alguna manera (los programas BASIC que hemos salvado y los ficheros de datos que hemos construido se escribían en la cinta sin necesidad de que supiéramos

exactamente en qué lugar de la memoria del ordenador estaban los datos a almacenar). Sin embargo, cuando queremos salvar programas en código máquina, o bloques de bytes, necesitamos saber dos cosas acerca de ellos.

i La posición en memoria del primer byte a salvar: DIRECCION DE COMIENZO.
ii La posición de memoria del último byte a salvar: DIRECCION FINAL.

Los comandos utilizados para escribir y leer bytes de la cinta son BSAVE y BLOAD, respectivamente. Intente recordar las sentencias como Byte SAVE y Byte LOAD. La instrucción BSAVE puede tener dos o tres parámetros. Los parámetros esenciales son las direcciones de comienzo y de final. El tercer parámetro es la DIRECCION DE EJECUCION, y sólo es especificado si el fichero contiene un programa en código máquina. La dirección de ejecución es la dirección donde el programa es ejecutado, y proporciona información al ordenador en el momento de la carga del fichero. El comando

```
BSAVE "test",200,499
```

salva 299 bytes de memoria, empezando en la dirección 200 y acabando en la 499. Se puede pasar como parámetro un descriptor de periférico con el nombre de fichero, como "CAS:test"; pero, en el momento de escribir, el único periférico abierto es el cassette. El nombre de fichero, como cabe esperar, puede reemplazarse por una variable de cadena. Las direcciones de comienzo y de fin también pueden ser reemplazadas por variables numéricas, como demuestra el comando siguiente.

```
comienzo=200
fin=499
BSAVE "test",comienzo,fin
```

salvará los bytes entre inicio y fin. Aquí no podríamos usar *end* como nombre de variable porque es una sentencia BASIC. Si se precisa especificar una dirección de ejecución habrá que pasar al tercer parámetro, como vemos a continuación:

```
BSAVE "juego",49800,50000,49820
```

Este comando salvará los bytes comprendidos entre 49800 y 50000, con una dirección de ejecución de 49820. Si no se especifica la dirección de ejecución, se supone que es la dirección de comienzo.

Para cargar de nuevo los bytes salvados con BSAVE, se utiliza el comando BLOAD. En su forma más simple, BLOAD se usa de la manera siguiente:

```
BLOAD, "CAS:"
```

Cuando sólo se utiliza el descriptor de periférico como en el ejemplo anterior, el comando cargará el siguiente fichero de tipo apropiado encontrado en la cinta. El comando

```
BLOAD "test"
```

cargará el fichero “test” de la cinta. En ambos ejemplos el fichero se carga en la dirección donde estaba almacenado originalmente. Si detrás del nombre de fichero se escribe la letra “R”, como en

```
BLOAD "juego",R
```

el fichero es cargado y ejecutado seguidamente (la dirección de ejecución es la especificada en la instrucción de archivado del fichero). Hay un parámetro más que podemos usar con el comando BLOAD. Se llama “OFFSET”, y nos permite cargar el fichero en el ordenador en una dirección diferente de donde estaba almacenado al ser salvado.

```
BLOAD "juego",R,200
```

cargará el fichero en (inicio+200). A cualquier dirección de ejecución que se especificase cuando se escribió el fichero también se le sumarán 200.

Hasta aquí las ventajas del manejo de cinta en BASIC MSX. Antes de terminar este capítulo, unas anotaciones acerca de un par de puntos que pueden ser útiles en determinadas ocasiones. Primeramente nos referiremos a la función EOF(*n*), donde *n* es un número de fichero. Esta función retorna el valor −1 si se ha encontrado el fin “fin del fichero” *(end of file) n*, y el valor 0 si no se ha encontrado. Si diseña las rutinas que manejan ficheros de tal manera que usted conozca cuántos registros más hay en el fichero, probablemente nunca use esta función. Si intenta leer un dato de un fichero que no contiene más, se genera un error. Comprobando el fichero con la función EOF(*n*) antes de realizar cualquier lectura prevendrá este suceso. Esta función es particularmente útil si el fichero a leer es de longitud desconocida.

Finalmente nos referimos a los errores que aparecen algunas veces trabajando con cintas.

Errores del sistema de cinta

Bad File Name (nombre de fichero incorrecto). Número de error 56.
Este error es causado por un nombre de fichero incorrecto en un comando LOAD, SAVE, etc.

Input Past End (lectura después del final). Número de error 55.
Es causado por intentar leer datos de un fichero que está vacío, o del que se han leído ya todos los datos.

File Already Open (fichero ya abierto). Número de error 54.
Se ha emitido una sentencia OPEN para un fichero que estaba ya abierto.

File Not Open (fichero sin abrir). Número de error 59.

Se ha emitido una sentencia PRINT # o INPUT # para un fichero que no está abierto. Debe abrirse el fichero antes de utilizar estos comandos.

Direct Statement in File (sentencia directa en un fichero). Número de error 57.

Este error ocurre ocasionalmente si se encuentra un comando directo durante la carga de un fichero en formato ASCII. El proceso es interrumpido inmediatamente.

Bad File Number (número de fichero incorrecto). Número de error 52.

Ocurre cuando un comando o sentencia se refiere a un fichero con un número fuera del rango fijado por MAXFILES, o se intenta acceder a un fichero que no está abierto. También se genera si intenta ejecutar un comando INPUT que ha sido abierto para salida o viceversa. Así es un mensaje de error general que recoge el resto de las posibilidades.

Device I/O Error (error E/S de periféricos). Número de error 19.

Este error se produce cuando ocurre algo bastante drástico: ejemplos típicos son: el usuario pulsando CTRL-STOP durante una operación de cinta, un fragmento de datos mal grabado, ruido electrónico a la entrada de cinta, que alguien desconecte la clavija de los auriculares del cassette durante una operación de carga. Esto puede ocurrir también con la impresora, si está conectada, o con la pantalla, si algo falla en el interior del ordenador.

NOTAS DEL CAPITULO

1.—Byte de menos valor: Se refiere al byte menos significativo.

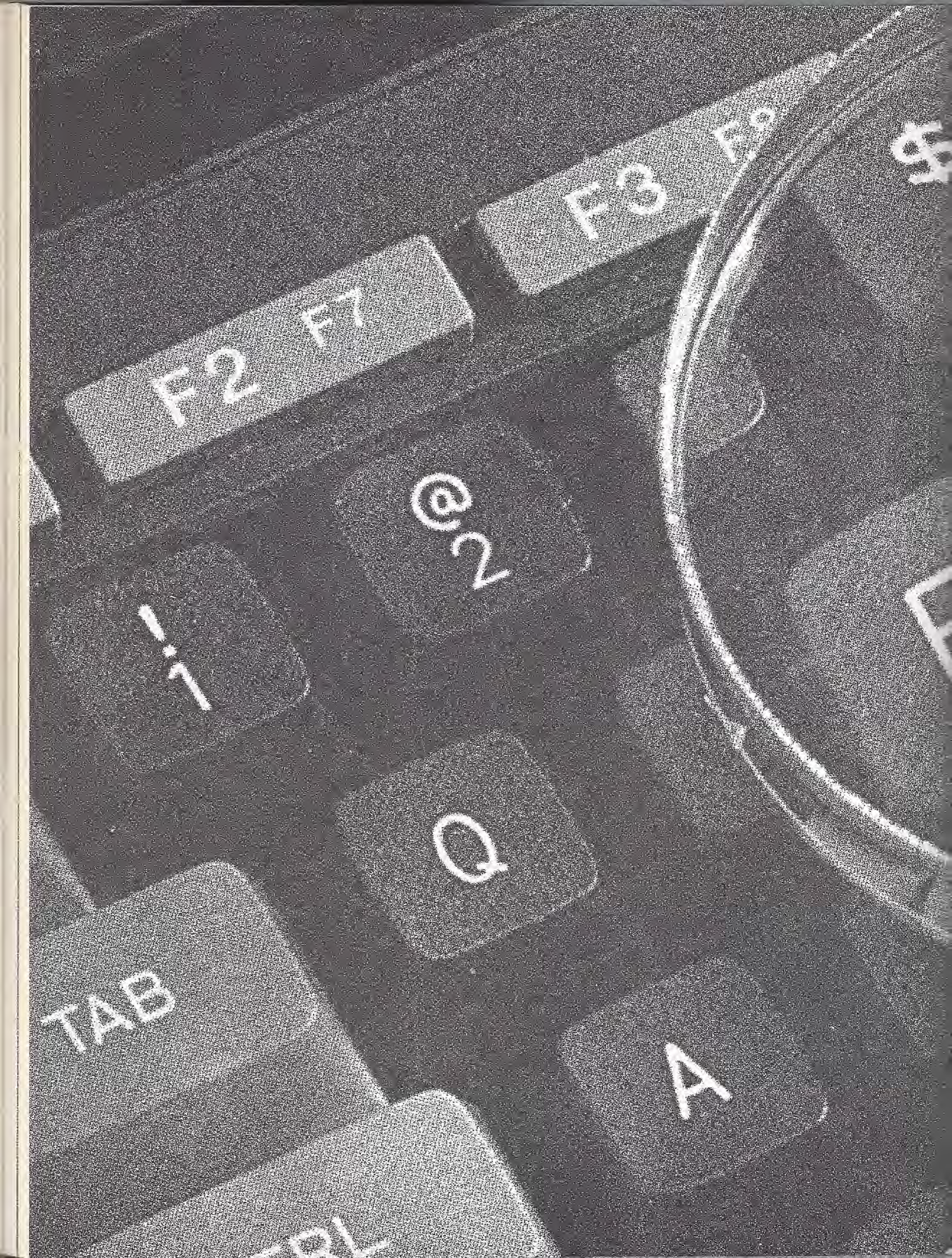
2.—CRT es abreviatura de *cathode ray tube* (tubo de rayos catódicos) y se refiere a la pantalla de video.

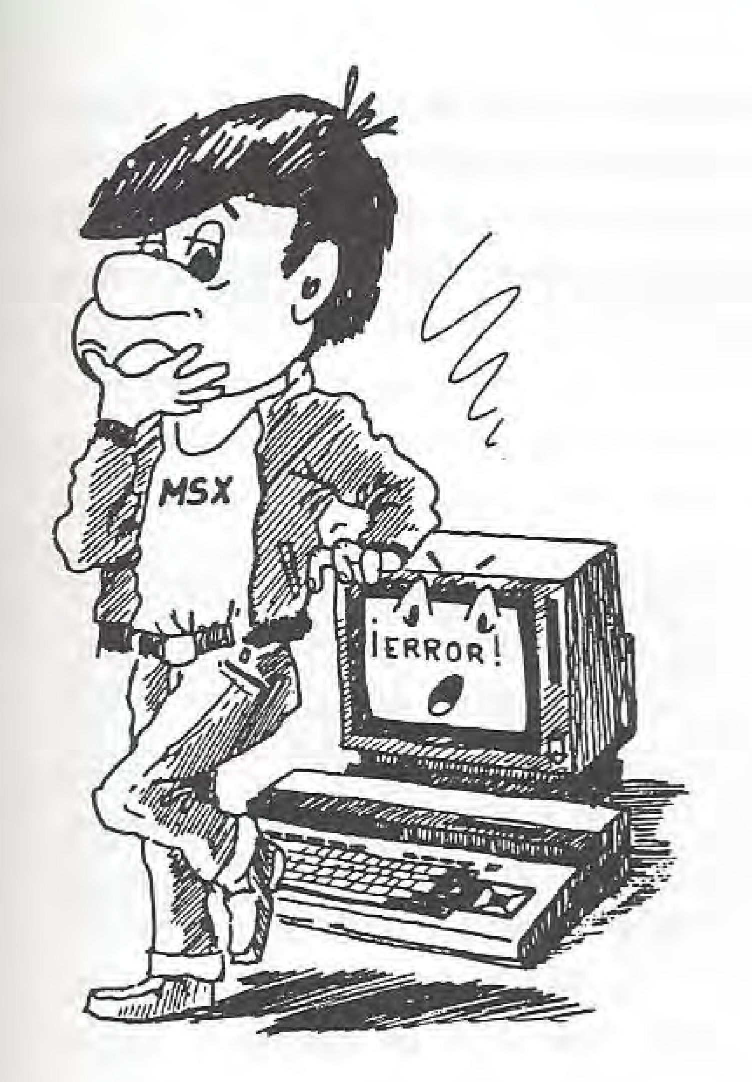
3.—Los ficheros del mismo tipo que el buscado, pero de distinto nombre, se indican en la pantalla de la manera siguiente:

Skip:"nombre del fichero"

Skip en español quiere decir "salto".

4.—BUFFER: Memoria intermedia. Almacenamiento provisional de los datos transmitidos de un dispositivo a otro. Se trata de una zona de memoria que sirve de almacenamiento intermedio entre la memoria central y un periférico.


F2 F7
F3
@
2
!
1
Q
TAB
A



5 Los comandos ON

Mientras se escribía este libro surgieron frecuentemente varios pequeños problemas que necesitaban atención. Cuando ocurría esto, dejaba la máquina de escribir, anotando aquello que debía escribirse al reiniciar el trabajo, resolvía el problema y continuaba con el libro como si nada hubiera pasado. Esto no es una disertación acerca de los rigores de la vida de un escritor, pero sí un ejemplo de una INTERRUPCION... En el mundo de los ordenadores una "interrupción" es una señal aplicada a la CPU del ordenador para informarla de que un circuito, en algún lugar del ordenador, necesita atención inmediata. La CPU termina su tarea, almacena los contenidos de sus registros internos, y sólo entonces realiza el trabajo requerido. A esto se le llama "tratar la interrupción", y, una vez sea completado el trabajo, la CPU reanuda su labor anterior.

El ejemplo de interrupción que probablemente sea más familiar para usted en los ordenadores MSX es la acción de la tecla STOP. Siempre que esto ocurre, el pulsar esta tecla —especialmente cuando lo hace a la vez que pulsa la tecla CTRL— provocará que se interrumpa la ejecución del programa. Otras interrupciones en el sistema MSX leen el teclado o ayudan a mantener funcionando la televisión o el monitor. Lo fundamental en las interrupciones es que, sea cual sea el trabajo que esté haciendo el ordenador, la aparición de una interrupción eventual siempre causa que la CPU vaya a hacer su cometido, regrese y continúe como si no hubiera pasado nada.

Esto es muy similar al comportamiento de los ordenadores MSX cuando utilizamos los comandos ON en programas BASIC. Ciertos acontecimientos, tales como un error, la pulsación de una tecla de función o la colisión de 2 *sprites,* motivarán que el control del programa pase de la línea ejecutada en ese momento a una rutina

especial, que "trata" la "interrupción" BASIC que causó el acontecimiento. Estos comandos son muy poderosos; por ello ocupan un capítulo por sí solos. Comencemos examinando alguna de estas instrucciones, empezando con la sentencia ON ERROR, una orden que nos permite hacer nuestros programas mucho más fáciles de usar para el resto de la gente, y para nosotros mismos.

ON ERROR

Todos nosotros nos despreocupamos del comportamiento normal del ordenador cuando ocurre un error; se interrumpe la ejecución del programa, aparece un mensaje de error, o INFORME impreso en la pantalla, y las variables del sistema ERR y ERL contienen detalles acerca del tipo de error y en qué lugar del programa ocurrió. Vimos en el capítulo 2 que todos los errores tienen códigos numéricos asociados; ahora veremos cómo podemos utilizar estos códigos para conseguir que el ordenador reanude la ejecución con la mínima confusión cuando ocurre un error. Esto es muy importante si alguien que no es experto en MSX está ejecutando su programa. El repentino *beep* (pitido que emite el ordenador) y el mensaje *Input Past End* motivará que mucha gente piense que los ordenadores son un peligro. La orden ON ERROR pasa el control a un cierto número de línea cuando ocurre un error. La sintaxis completa es

```
ON ERROR GOTO n
```

donde *n* es un número de línea existente. Así, el comando

```
ON ERROR GOTO 3000
```

al principio de un programa BASIC hará que el control del programa pase a la línea 3000 siempre que ocurra un error. Las sentencias del programa a partir de la línea 3000, que se ocupan de los errores, se llaman en conjunto RUTINA DE TRATAMIENTO DE ERROR. Cuando usamos el comando ON ERROR se dice que estamos facilitando el tratamiento de error.

Veamos un ejemplo de tratamiento de error funcionando. Teclee la instrucción NEW, e introduzca el siguiente programa. Veremos con más detalle los modos de pantalla en el capítulo dedicado al VDP.

```
10 SCREEN 1
20 ON ERROR GOTO 3000
30 FOR I=1 TO 30
40 PRINT I
50 NEXT I
60 GOTO 30
3000 SCREEN 0:PRINT "Numero de error "
;ERR;" ocurrido en la linea ";ERL
3010 END
```

Antes de que veamos el programa operando como una trampa de errores, ejecútelo y asegúrese de que funciona. Si el mensaje de la sentencia PRINT en la línea 3000 es impreso en la pantalla, ¡ha provocado un error de programación! Suponiendo que esté todo en orden, el programa empezará a imprimir los números mediante el bucle FOR ... NEXT. Pulse CTRL-STOP para parar el programa. Este error no es detectado por el comando ON ERROR, pero sí por un comando que veremos después en este capítulo. Sin embargo, mientras la máquina esté en modo directo, teclee algo sin sentido, algo que normalmente generaría un error de sintaxis o cualquier otro error. Con un poco de suerte, entrará desde el modo directo, por este método, en el tratamiento de errores. El mensaje "Ooops..." será impreso, junto con el número de error y el número 65535, en vez de un número de línea. El número de error dependerá del error que haya causado. Observe que el código del número de error en la rutina del tratamiento de errores se obtiene de la variable del sistema ERR, y que el número de línea donde ocurrió el error proviene de ERL. Para ver cómo opera el tratamiento de errores desde dentro del programa reemplace la línea 40 con

```
LPRINT I
```

Se supone que no tiene la impresora conectada. Ejecute el programa. Seguramente, el ordenador no podrá ejecutar dicho programa debido a que la impresora no está conectada. Si ahora pulsamos CTRL-STOP, actuará el tratamiento de error con ERR = 19, porque se ha generado un error I/O. También es posible entrar en el tratamiento de errores usando el comando ERROR, ya sea en modo directo o desde dentro de un programa.

¿Qué demuestra esta simple rutina? Nos ha enseñado que las variables del sistema pueden ser tratadas como variables BASIC normales; pueden ser impresas como parte de la lista de expresiones de una sentencia PRINT, y pueden ser usadas en sentencias IF ... THEN ... ELSE, como enseguida veremos. También hemos visto las distintas maneras de pasar el control del programa a una rutina de tratamiento; la aparición de un error en modo directo, durante la ejecución de un programa, o el uso del comando ERROR. Sin embargo, esta simple demostración no nos ayuda a recuperar el control del programa. Para ello, necesitamos ver otro comando, llamado RESUME.

Cambie el END de la línea 3010 por RESUME y repita lo experimentado anteriormente. Verá algo bastante interesante, pero de efectos no demasiado útiles. Siempre que una rutina de tratamiento de error encuentra una sentencia RESUME, el ordenador devuelve el control a la sentencia que causó el error. Así, cuando el programa entra en la rutina de tratamiento de error debido a uno de sintaxis, la máquina, en lugar de informar del hecho y limitarse a parar, intenta interpretar de nuevo la sentencia errónea, lo cual provocará que el ordenador entre eventualmente en un bucle sin fin, y la única manera de salir de él es pulsar CTRL-STOP. Una situación similar tendrá lugar con los otros errores. Por tanto, el comando RESUME, por sí mismo, no es muy útil. No obstante, tiene otras aplicaciones más provechosas, como veremos a continuación:

```
3010 RESUME NEXT
```

es el siguiente comando a experimentar. Ahora, una vez ejecutada la rutina de tratamiento de error, el comando RESUME NEXT pasa el control del programa a la siguiente sentencia, que aparece después de la que provocó el error.

La última aplicación de la instrucción RESUME es la más útil. Generalmente requerimos rutinas de tratamiento de errores que permitan recomenzar el programa o que vuelvan a ejecutar una parte del mismo. Por ejemplo, si se genera un error I/O, probablemente queramos volver a intentar ejecutar la parte del programa que causó el error, habiéndole sido mostrada al usuario con anterioridad la naturaleza del error. Por ejemplo, se puede pedir al usuario que compruebe el cassette o la impresora antes de intentarlo de nuevo. La instrucción que nos permite regresar a un cierto número de línea después de ejecutar una rutina de tratamiento de error es

```
RESUME n
```

donde *n* es un número de línea que existe en el programa. Veamos el ejemplo siguiente, proveniente de un programa que usa ficheros en cinta. Se consideran dos errores principales; un *Device I/O Error* o un *Bad File Name*, que tienen los números 19 y 56, respectivamente.

```
3000 REM Rutina de tratamiento de error
3010 IF ERR=19 THEN RESUME 10
3020 IF ERR=56 THEN RESUME 20
3030 ON ERROR GOTO 0
```

La línea 10 del programa sería un menú, de tal manera que el usuario podría volver a intentar la operación que causó el error de I/O. La línea 20 le pediría un nombre de fichero; de esta manera, al producirse un error de nombre de fichero incorrecto, el usuario es preguntado de nuevo. La instrucción ON ERROR GOTO 0 de la línea 3030 haría que no todos los errores provocasen un salto dentro del programa y permitiría un informe sobre su naturaleza y del lugar donde aparecieron. En este caso, cualquier error distinto de los de números 19 y 56 conllevaría un tratamiento normal. Una vez se ha entrado en la rutina de tratamiento de error, la instrucción ON ERROR queda sin efecto hasta que se ejecute una instrucción RESUME. Cuando esto ocurre, la sentencia ON ERROR es reactivada.

En este último método, RESUME siempre debe ir seguido de un número de línea, que no puede ser una variable o una expresión que dé un número de línea correcto, sino que debe ser una constante. Las rutinas de tratamiento de error pueden incluir llamadas a subrutinas; los errores que se generan en esas rutinas darán su mensaje de error característico, ya que estas rutinas quedan desactivadas hasta que aparezca RESUME.

ERROR

Las rutinas de tratamiento de error trabajarán también con el comando BASIC ERROR *n*, permitiendo así al usuario fijar sus propios errores, que pueden ser tra-

tados como cualquier otro error BASIC. De nuevo, ERR y ERL contendrán los valores apropiados que definen el error.

Múltiples ON ERROR

Un programa BASIC puede contener más de una sentencia ON ERROR. Cada una de éstas prevalece sobre la última ejecutada, y por eso se pueden dirigir los errores a diferentes rutinas de tratamiento de error, según el lugar del programa en que éste se produjo. Por ejemplo:

```
10 ON ERROR GOTO 2000
20 PRINT "Hola"
30 hdjfd:REM Error deliberado
40 ON ERROR GOTO 3000
50 hbvasjhg:REM Error deliberado
60 PRINT "Hola"
70 END
2000 REM Primera rutina de error
2010 PRINT "2000"
2020 RESUME NEXT
3000 REM Segunda rutina de error
3010 PRINT "3000"
3020 RESUME NEXT
```

El primer error deliberado del programa pasará a la rutina de tratamiento de error de la línea 2000. La sentencia ON ERROR GOTO de la línea 40, sin embargo, direcciona el error siguiente a la línea 3000.

POSIBLES PROBLEMAS

Como ocurría con las subrutinas, es de vital importancia que el programa no entre accidentalmente en la rutina de tratamiento de error de una manera incorrecta. GOSUB es el método apropiado de entrada en una subrutina y el único que permite que un programa pueda ser capaz de entrar en la rutina de tratamiento de error cuando se genera uno. Si se entra en una de estas rutinas sin haberse producido un error, la ejecución de la instrucción RESUME, en cualquiera de sus formas, producirá un mensaje de error. Por tanto, es imposible que un programa entre accidentalmente en una rutina de tratamiento de error. Las rutinas deben estar separadas del resto del programa por las instrucciones END o STOP.

Finalmente, mencionaremos el comando LIST., un caso especial de LIST. Ya se ha comentado que LIST ha de preceder a una constante, no a una variable, aunque ésta sea del sistema como ERL. LIST., cuando aparece en una rutina de tra-

tamiento de error, interrumpirá el programa listando la línea de éste donde ocurrió el error. LIST. puede tratarse como si fuera el comando LIST ERL. Esto es útil para el depurado de programas. También puede usarse en modo directo.

Comando ON KEY

Este comando, cuya sintaxis completa es

```
ON KEY GOSUB 100,200,300 ...
```

nos permite definir una serie de números de línea a los que se pasará el control del programa cada vez que se pulse una de las teclas de función[1]. El control se pasará en forma de llamada a una subrutina; por ello, estas rutinas han de terminar con RETURN. En el ejemplo anterior se pasará el control del programa a la subrutina de la línea 200 siempre que se pulse la tecla F1, mientras el programa está rodando. Este INCIDENTE, que así es como se llama, solamente se tratará de esta manera si la tecla de función implicada ha sido "activada" por un comando KEY(*n*) ON, donde *n* es el número de la tecla de función.

Veamos un ejemplo de tratamiento de la tecla de función F1 para saltar a una subrutina, siempre que sea pulsada. No se preocupe por los comandos SCREEN ni por el resto de los comandos gráficos; veremos ambos en el capítulo 6.

```
10 KEY(1) ON
20 ON KEY GOSUB 1000
30 SCREEN 1
40 FOR I=1 TO 10
50 FOR J=1 TO 100:NEXT
60 PRINT I
70 NEXT
80 GOTO 30
90 END
1000 SCREEN 3
1010 OPEN "GRP:" FOR OUTPUT AS #1
1020 PRESET (0,20),2
1030 PRINT #1,"Eso era F1"
1040 CLOSE #1
1050 TIME =0
1060 IF TIME<200 THEN GOTO 1060
1070 RETURN
```

La línea 10 activa la tecla que utilizaremos cuando pretendamos entrar en una rutina de tratamiento, en este caso F1, y la línea 20 indica la línea donde va a

pasar el control cuando se pulse la tecla (en este ejemplo, la línea 1000). El número de tecla de la línea 10 no tiene por qué ser una constante; puede ser una variable o una expresión que retorne un valor apropiado. Así podemos sustituir legalmente las líneas siguientes en el programa anterior:

```
5 n=1
10 KEY (n) ON
```

El comando KEY ON puede ir antes o después de la sentencia ON KEY GOSUB. En el ejemplo anterior, únicamente hemos activado la tecla 1. Suponga que sólo queremos usar la tecla F5 para provocar el salto; en este caso se coloca una coma en la sentencia ON KEY GOSUB por cada tecla que esté inactiva. Así, para hacer operar la tecla 5 en lugar de la 1 en el último programa ejemplificado, cambie las líneas 10 y 20 por las dos siguientes:

```
10 KEY (5) ON
20 ON KEY GOSUB ,,,,1000
```

De una manera similar, las líneas

```
10 KEY (6) ON
20 ON KEY GOSUB ,,,,,1000
```

harán que la tecla 6 provoque el salto a la subrutina de la línea 1000. Si queremos que el salto a distintas subrutinas se efectúe por distintas teclas de función, es necesario emitir un comando KEY ON por cada tecla que se desee activar, y también definir un número de línea por cada una de ellas en la sentencia ON KEY GOSUB. Así, las líneas

```
10 ON KEY GOSUB 2000,,3000,4000
20 KEY(1) ON
30 KEY(3) ON
40 KEY(4) ON
```

activarán las teclas 1, 3 y 4, de tal manera que, cuando el programa está en marcha, al pulsar la tecla 1 el control del programa pasará a la línea 2000; pulsando la tecla 3, a la 3000, y pulsando la 4, a la 4.000. Si la tecla es activada por la sentencia KEY ON, pero no se da un número de línea en ON KEY GOSUB, cualquier presión sobre la tecla sencillamente será ignorada.

Cuando tenemos que activar varias teclas con la sentencia KEY ON, es evidente que usar varias instrucciones KEY ON probablemente NO es el método más eficiente para realizar esta función. Mi método favorito es usar bucles FOR ... NEXT o sentencias DATA. Por ejemplo, observe cómo se activan las teclas 1 a 6:

```
10 FOR I=1 TO 6
20 KEY(I) ON
30 NEXT
```

Si son seis las teclas a activar, pero no en ese mismo orden, usaremos una sentencia DATA y un bucle FOR ... NEXT para leer lo que contiene la sentencia, como en el programa siguiente:

```
10 FOR I=1 TO 6
20 READ n
30 KEY(n) ON
40 NEXT
1000 DATA 1,2,4,6,7,8
```

Este último programa activa las teclas 1,2,4,6,7 y 8, algo que encerraría cierta dificultad para hacerlo correctamente utilizando sólo bucles FOR ... NEXT.

Si deseamos desactivar una tecla de función que ha sido activada en un programa, usamos la instrucción KEY (*n*) OFF. Esto resulta útil cuando queremos que la tecla de función esté activa en una parte del programa e inactiva en otras. así, el comando

```
400 KEY(2) OFF
```

asegurará que pulsar la tecla F2 no tiene efecto si ha sido ejecutada esta instrucción. Desde luego, si quisiéramos activar de nuevo la tecla de función podríamos usar un comando KEY ON para hacerlo.

KEY (*n*) STOP tiene efectos similares a los del comando KEY OFF; sin embargo, si pulsamos la tecla de función, ésta se ignora después de un comando KEY OFF; si estuviera activo KEY STOP este hecho quedaría registrado. De tal manera que tan pronto como se emitiera un comando KEY ON para esa tecla en concreto, el ordenador recordaría que ha sido pulsada durante el período KEY STOP, y pasaría el control a la línea de programa apropiada. Si la tecla de función implicada no ha sido pulsada durante el período STOP, no ocurre nada.

Cuando se entra en una rutina de este tipo, el ordenador emite un comando KEY STOP para esa tecla de función particular. Cuando se encuentra un RETURN se vuelve a activar la tecla. Así, si se presiona la tecla de función durante la ejecución de su subrutina, la rutina será ejecutada una segunda vez tan pronto como aparezca el correspondiente RETURN.

Si al utilizar el comando ON KEY GOSUB necesita recordar qué función tiene cada tecla, no olvide que puede definir una cadena para cada tecla de función. Esta se escribirá generalmente en la línea 24 de la pantalla. La naturaleza del texto contenido en la tecla de función no diferencia la manera en que el ordenador responde a la operación de ON KEY GOSUB cuando se pulsa la tecla. Recuerde que si esta línea 24 de la pantalla le molesta, el comando

```
KEY OFF
```

sin número de tecla la borrará de la pantalla. Para volver a editarla pulse KEY ON.

El comando ON que veremos a continuación tiene que ver con el tratamiento de interrupciones, esta vez causado por la tecla STOP.

ON STOP GOSUB n

El método principal de interrupción de un programa en los ordenadores MSX es pulsar simultáneamente las teclas CTRL y STOP (generalmente, esta acción se conoce como un comando Control-Stop o, abreviado, CTRL-STOP). El programa interrumpirá su ejecución inmediatamente. Esto, sin embargo, no es siempre deseable, especialmente si el programa va a ser usado por alguien que usted no desea que pueda ver el listado del mismo. ON STOP GOSUB *n* le permite evitar la función de interrupción dada por CTRL-STOP en sus programas. No afecta al uso aislado de la tecla STOP, que habitualmente produce una pausa en la ejecución del programa, y tampoco impide la acción del comando BASIC STOP desde dentro del mismo. La sentencia ON STOP GOSUB *n* sólo actúa cuando se pulsa CTRL-STOP estando un programa BASIC en curso de ejecución. Cuando está activa, este hecho provocará que se ejecute la subrutina que comienza en la línea indicada en la instrucción ON STOP GOSUB. Por medio de la sentencia RETURN se devuelve el control al programa. Como todas las subrutinas que hemos visto en este capítulo, con la excepción de las de ON ERROR que acababan con un comando RESUME, ésta termina con RETURN. El comando ON STOP GOSUB tiene en común con el resto de las instrucciones estudiadas en este capítulo que debe ser activado usando

```
STOP ON
```

que puede ser emitido antes o después de ON STOP GOSUB. Por eso, las dos líneas

```
10 ON STOP GOSUB 5000
20 STOP ON
```

provocarán que el control del programa pase a la línea 5000 siempre que se dé el incidente CTRL-STOP. Como ejemplo de lo que puede hacerse con este comando, el fragmento de programa siguiente desactivará por completo la operación de CTRL-STOP, y esta combinación de teclas no interrumpirá el programa cuando esté funcionando:

```
1 ON STOP GOSUB 20000
2 STOP ON
......
......
20000 RETURN
```

Si inutiliza la tecla STOP de esta manera en un programa, ¡cuidado! Es posible que el ordenador entre en un bucle infinito, y la única manera de sacarlo es haciendo un *reset* en la máquina, con la consiguiente pérdida del programa.

El comando STOP OFF desactiva la instrucción CTRL-STOP. El STOP STOP realiza la misma función, pero si se presiona CTRL-STOP se recuerda el incidente y, en consecuencia, se actúa tan pronto como aparezca una sentencia STOP ON.

Hay una ocasión en la que un programa BASIC protegido de este modo puede ser interrumpido usando la función CTRL-STOP. Es cuando se ha entrado en una rutina de tratamiento de error; hasta que se llega a la sentencia RESUME, todos los comandos ON son desactivados.

Cuando el ordenador ha ejecutado una rutina de tratamiento de interrupción, la sentencia RETURN también ejecuta un comando STOP ON. Esto se debe a que, durante el manejo de la rutina, el ordenador actúa como si se hubiese emitido un comando STOP STOP. Si escribe un comando STOP OFF dentro de la rutina, la instrucción CTRL-STOP no es desactivada, y la sentencia RETURN, así como cualquier subisiguiente CTRL-STOP, tendrá el efecto normal.

ON SPRITE GOSUB n

Este comando lo estudiaremos cuando tengamos un mejor conocimiento de los *sprites* disponibles en BASIC MSX. Esta cuestión se tratará completamente en el siguiente capígulo, donde también veremos este comando.

ON INTERVAL = tiempo GOSUB n

Este comando nos permite programar una interrupción siempre que transcurra un intervalo de tiempo, ejecutando una subrutina que comienza en la línea *n*, y al regresar de la subrutina proseguir hasta que el intervalo de tiempo transcurre de nuevo; entonces se repetirá el proceso. Para implementar este comando, el ordenador hace uso de una interrupción generada por el VDP, que permite a la CPU ejecutar algunas rutinas contenidas en ROM que leen el teclado y actualizan el valor de la variable TIME. El número de veces por segundo que esto ocurre depende del tipo de sistema MSX que tengamos. Si es un ordenador que utiliza como pantalla una televisión europea, esta interrupción ocurre cincuenta veces por segundo. Si la máquina fue comercializada para el mercado japonés y, por tanto, no puede utilizar una televisión europea, la interrupción ocurre sesenta veces por segundo. El ordenador cuenta el número de interrupciones y hace uso de ello cuando utilizamos el comando ON INTERVAL.

El valor que debe darse a la instrucción se obtiene multiplicando el número de veces por segundo que se genera la interrupción por el tiempo de demora en segundos. En Europa, el valor del parámetro de tiempo viene dado por

```
tiempo=tiempo de demora*50
```

Así, para un tiempo de demora de cinco segundos entre ejecuciones de la subrutina especificada en el comando ON INTERVAL, la instrucción usada sería:

```
10 ON INTERVAL=250 GOSUB 1000
```

Para activar el comando ON INTERVAL ha de emitirse una sentencia INTERVAL ON de una manera similar a los comandos KEY (*n*). De nuevo, la subrutina requerida debe terminar en RETURN.

El programa listado a continuación hace uso del comando ON INTERVAL para dar un "reloj de alarma". Una vez ha sido ejecutado el comando INTERVAL ON, el ordenador entra en la subrutina de la línea 1000 cada diez segundos. La frecuencia de entrada en la subrutina puede variarse alterando el parámetro de tiempo de la línea 10.

```
5 SCREEN 0
10 ON INTERVAL=500 GOSUB 1000
20 INTERVAL ON
30 FOR I=1 TO 1000000
40 PRINT I
50 NEXT I
60 END
1000 SCREEN 3
1010 OPEN "GRP:" FOR OUTPUT AS #1
1020 PRESET (0,40)
1030 PRINT #1,"-Despierta!"
1040 BEEP:BEEP
1050 CLOSE #1
1060 FOR T=1 TO 2000:NEXT T
1070 SCREEN 0
1080 RETURN
```

Tan pronto como se emite el comando INTERVAL ON de la línea 20, la rutina de la línea 1000 se ejecuta cada vez que termina el tiempo de demora; en este caso, por ejemplo, la rutina será ejecutada a los 10, 20, 30... etc. segundos. Este tiempo incluye el tiempo que se tarda en ejecutar la rutina. Así, si la rutina tarda en ejecutarse tres segundos y el parámetro de tiempo del comando ON INTERVAL está fijado a 500, se hará una reentrada en la rutina a los siete segundos de haberla acabado. Si desea solamente que actúe una vez el tiempo de demora, una instrucción INTERVAL OFF en la subrutina asegurará que ocurra esto. Al entrar en la subrutina se ejecuta un comando INTERVAL STOP. Esto tiene efectos similares a STOP STOP. Si transcurre otro intervalo de demora mientras se ejecuta la rutina, tan pronto como se ejecute el comando RETURN de la línea 1080 ésta se volverá a ejecutar. Puede verlo en acción reduciendo el parámetro de tiempo en la línea 10 del programa de demostración a, digamos, 10. Es un ejemplo simple, pero si da un valor bajo al parámetro de tiempo, el tiempo de ejecución de de la rutina tiene que ser pequeño siempre que el programa tenga que hacer alguna otra cosa aparte de repetir esta rutina. Por supuesto, INTERVAL OFF e INTERVAL STOP pueden usarse en otras partes del programa para impedir la acción de ON INTERVAL

cuando éste no es necesario. Dentro de las restricciones mencionadas anteriormente, la rutina puede ejecutar cualquier comando que desee. Como ocurre con los otros comandos ON, el programa proseguirá su ejecución normal, como si nada hubiera pasado, al llegar a la instrucción RETURN.

ON STRIG

Este comando se usa con el *joystick;* por tanto, lo veremos en el capítulo 7.

* * *

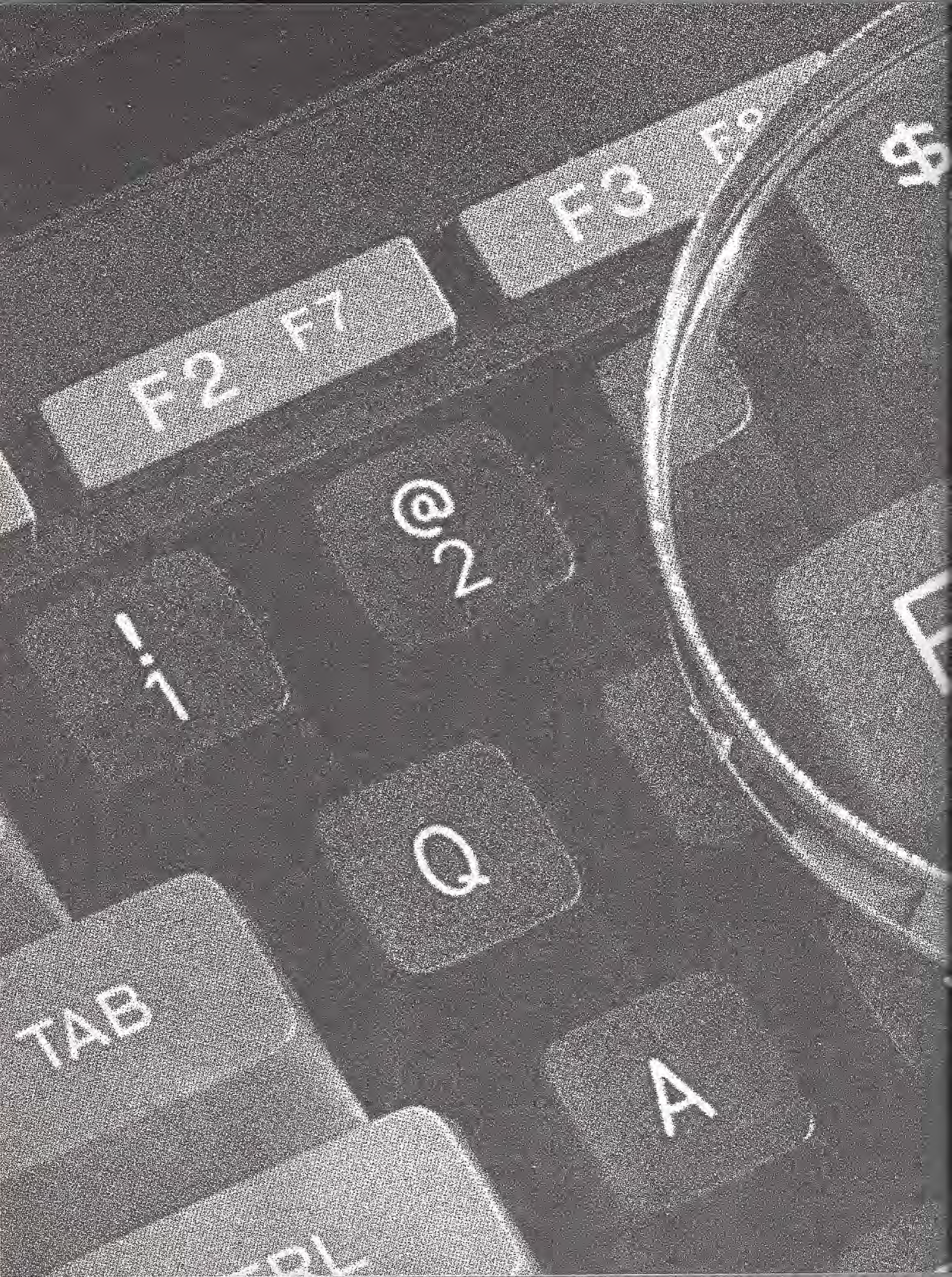
Aquí terminan los comandos ON. La capacidad de ON INTERVAL para ejecutar regularmente una subrutina se usará en el siguiente capítulo cuando veamos los comandos gráficos y de *sprites,* así como el procesador de *display* de video.

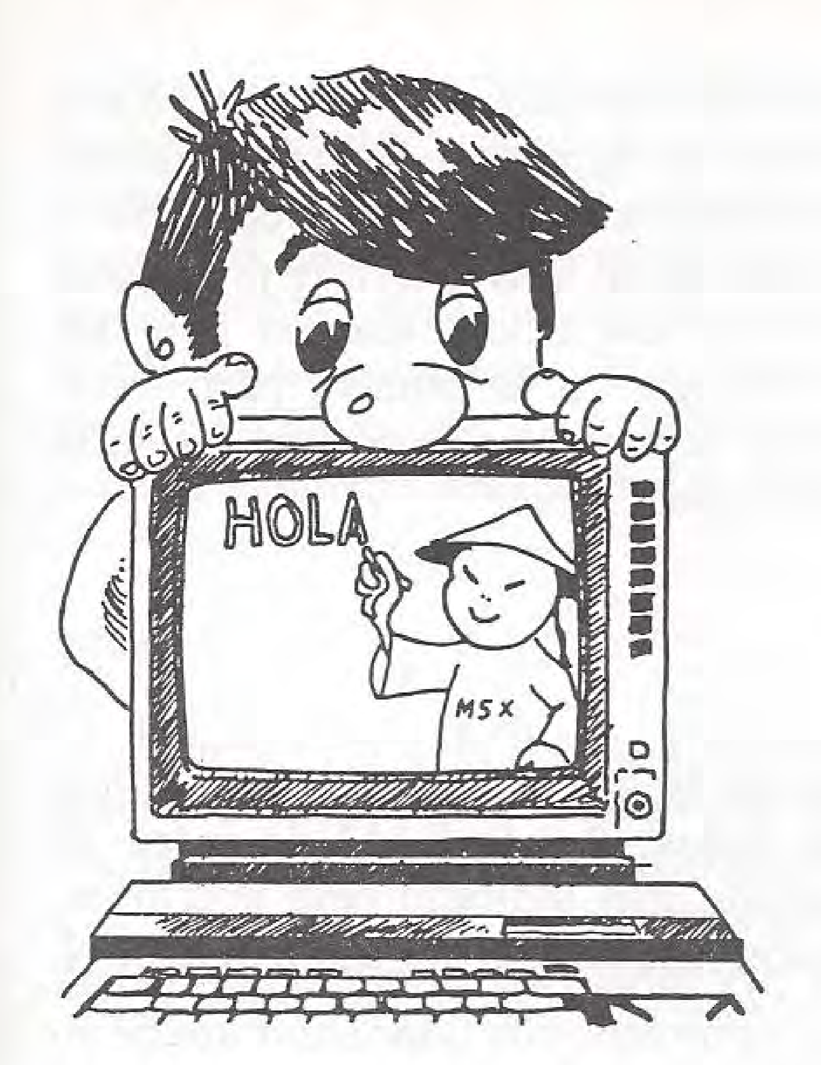
NOTAS DEL CAPITULO

1.—TECLAS DE FUNCION: Son una serie de teclas situadas en la parte superior del teclado, que generalmente están numeradas del 1 al 10. Cuando conecta el ordenador, contienen cadenas de comandos BASIC y pueden redefinirse por el usuario. Estos comandos BASIC iniciales son:

F1	color
F2	auto [10,10]
F3	goto
F4	list
F5	run+CHR$(13)
F6	color 15, 4, 4 +CHR$(13)
F7	cload"
F8	cont+CHR$(13)
F9	list.+CHR$(13)
F10	CLS+run+CHR$(13)

Para obtener de F6 a F10 hay que pulsar *Shift* y la tecla correspondiente. Estas teclas de función pueden utilizarse para controlar la ejecución del programa, como se explica en este mismo capítulo.


F2 F7
F3
$
@
2
!
1
Q
TAB
A



6 Procesador de video

El procesador de *display* de video o, más corto, el VDP es el circuito integrado del ordenador que proporciona al sistema MSX sus excelentes capacidades gráficas. Es uno de los *chips* que forma parte del MSX estándar, y por eso en este capítulo lo veremos detalladamente, empezando por alguna información general.

El circuito integrado usado en la especificación MSX es el TMS 9918 o alguno equivalente. Controla el *display,* que en el sistema MSX puede ser un televisor o un monitor, y en total controla 16.384 bytes de memoria de video, conocida como VRAM. Hay disponibles dieciséis colores, como veremos después. El VDP está comunicado con la unidad central de proceso a través del sistema de *bus* del ordenador, y se pueden dar los siguientes tipos de transferencia de información entre la CPU y el procesador de video.

i. La CPU envía bytes a los REGISTROS del VDP.
ii. La CPU lee bytes de los registros del VDP.
iii. La CPU envía bytes a la RAM de video.
iv. La CPU lee bytes de la RAM de video.

Un registro es la mejor manera de ver un byte de RAM dentro del *chip* procesador de video, que controla la operación del VDP. No forma parte de la VRAM. Dentro del VDP hay ocho registros llamados SOLAMENTE DE ESCRITURA; la CPU Z-80 puede escribir bytes en estos registros, pero no puede leerlos. También hay un registro SOLAMENTE DE LECTURA, llamado REGISTRO DE ESTADO.

Los veremos con más detalle más adelante, en este mismo capítulo. La RAM de video contiene datos concernientes a los *sprites*[1], el *display* de pantalla, los colores en uso y muchas otras cosas. Accediendo directamente a la RAM de video o a los registros del VDP podemos incrementar en gran medida el poder de programación a nuestra disposición. La segunda parte del capítulo tendrá que ver con este método de usar el VDP desde el BASIC. Sin embargo, en la primera parte veremos los comandos BASIC disponibles sin entrar de lleno en las disposiciones de la RAM de video o de los registros del procesador de video.

Modos de pantalla

Un modo de pantalla es la manera particular de la disposición del *display*. Los ordenadores MSX tienen cuatro modos, usando todos ellos la RAM de video de diferente manera, constituyendo cada modo el método más indicado para una determinada aplicación. Los modos de pantalla se seleccionan usando el comando SCREEN, y son los siguientes:

Modo 0	Modo de texto
Modo 1	Modo de texto
Modo 2	Modo de gráficos de alta resolución
Modo 3	Modo de gráficos de baja resolución.

A continuación detallaremos sus peculiaridades y aplicaciones.

Modo 0

Es un modo de texto que visualiza en la pantalla los caracteres del teclado del ordenador, esto es, letras, números y algunos caracteres no alfanuméricos (caracteres especiales). La pantalla en este modo se compone de 24 líneas de 40 caracteres por línea; sin embargo, los microordenadores MSX disponibles actualmente en Inglaterra disponen de 24 líneas de 37 caracteres. Los caracteres actuables visualizados en este modo se almacenan en una parte de la RAM de video. Esto quiere decir que podemos alterar la forma de las letras impresas en la pantalla alterando directamente la VRAM. Más adelante veremos cómo hacerlo. En este modo podemos utilizar dos colores de los dieciséis disponibles, uno para los caracteres visualizados y otro para el fondo.

Modo 1

Nos da 24 líneas de 32 caracteres, pero las máquinas inglesas visualizan 24 líneas de 29 caracteres. En la especificación MSX se puede disponer de dos colores de los dieciséis disponibles, uno para los caracteres visualizados y otro para el fondo. Sin embargo, manipulando la VRAM directamente podemos modificarlo algo. Luego es posible modificar los caracteres de la VRAM.

Los comandos gráficos del BASIC MSX, como PSET, LINE o DRAW, generarán errores si intenta usarlos en un modo de texto. No obstante, se puede disponer de *sprites* en modo 1 (se explica más adelante en este mismo capítulo). Una cuestión a tener en cuenta es que se retornará al modo de texto siempre que ejecute la sentencia INPUT desde un modo gráfico, o cuando el ordenador ha terminado la ejecución de un programa. Dado que esto provoca la pérdida de todo lo que hubiera en la pantalla en ese momento, recuerde finalizar el programa con un bucle infinito si desea examinar los resultados de los comandos gráficos.

Modo 2

El modo 2 nos da acceso a los gráficos de alta resolución. Podemos tener dieciséis colores en la pantalla y también usar *sprites*. La resolución de la pantalla es 256 por 192 *pixels*. Un *pixel* es como un punto en la pantalla. Sin embargo, en cada grupo de 8 *pixels* en dirección horizontal sólo puede haber dos colores separados, uno en primer término y otro de fondo. Por eso no es posible tener una fila de 8 *pixels* en dirección horizontal con diferentes colores. En dirección vertical, no obstante, no hay esas limitaciones. Para 8 *pixels* verticales, en este modo puede haber ocho colores diferentes si es necesario.

Modo 3

Pantalla de gráficos de baja resolución, con dieciséis colores y *sprites*. La resolución de la pantalla en este modo es de 64 por 48 *pixels*, horizontales y verticales, respectivamente. Cada *pixel*, en cualquier posición de la pantalla, puede tener un color determinado. No hay problemas con la resolución horizontal, como en el modo 2.

Si ha experimentado con todos los modos gráficos de los ordenadores MSX, probablemente está enterado del hecho de que las instrucciones de impresión normales no funcionan en pantallas gráficas. Hay una manera de resolver este problema, que veremos pronto.

Impresión de texto

Estudiaremos primero los comandos disponibles en los modos de texto, con la excepción del control de *sprites*, que ocupa una sección aparte en este capítulo. Para ver los caracteres utilizables en modo de texto introduzca el siguiente programa:

```
10 SCREEN 0
20 FOR I=32 TO 255
30 PRINT CHR$(I);
40 NEXT I
50 END
```

La línea 10 selecciona el modo de pantalla 0. Si aquí quisiéramos usar otro modo, simplemente procederíamos a poner el número apropiado en lugar de 0. Con el programa se imprimen los caracteres con código ASCII entre 32 y 255. Observe cómo los caracteres se imprimen a todo lo ancho de la pantalla. Podemos, de hecho, variar el número de caracteres escritos por línea usando el comando WIDTH.

```
WIDTH 20
```

fijará la línea de pantalla en veinte caracteres. En modo 0, los valores correctos del parámetro del comando WIDTH están entre 1 y 40, y en modo 1, entre 1 y 32. Incluya la línea siguiente en el programa anterior:

```
15 WIDTH 15
```

Observe cómo este comando afecta también a la línea 24 de pantalla donde se visualizan las teclas de función. La anchura de línea establecida mediante este comando permanece constante aunque cambie de modo de pantalla. La única manera de que tome su valor original es usando otro comando WIDTH.

Cuando haya llenado de texto una pantalla, el comando CLS la borrará. Este comando también actúa en los modos gráficos.

```
LOCATE X,Y
```

El comando LOCATE nos facilita posicionar un texto sobre una pantalla de texto donde queramos. La sintaxis completa del comando es:

```
LOCATE X,Y,cursor
```

El parámetro cursor es opcional, y por defecto toma el valor 0. El comando asegura que la siguiente sentencia de impresión emitida por el sistema comenzará a escribir en la posición X,Y de la pantalla, siendo X la posición horizontal tomada desde la izquierda e Y la vertical tomada desde la parte superior. En el sistema MSX, el carácter situado más arriba y a la izquierda de la pantalla tiene las coordenadas 0,0. X crece de izquierda a derecha e Y de arriba abajo. El parámetro del cursor determina si éste ha de visualizarse en la pantalla detrás de la siguiente sentencia de impresión. Vea el siguiente ejemplo:

```
10 LOCATE 10,10,0:PRINT "#"
20 GOTO 20
```

imprimirá el "#", pero nada más. Sin embargo, si alteramos la línea 10

```
10 LOCATE 10,10,1:PRINT "#"
```

producirá que se imprima "#" seguido del cursor. Si el parámetro de cursor es igual a 1, se dice que éste está activado, y si es igual a 0, desactivado.

COLOR

Hasta ahora no hemos dejado de ver letras blancas sobre un fondo azul, que son los colores que presentan todos los ordenadores MSX al ser conectados. Cambiar el color del texto es fácil... ¡si podemos recordar cómo se dice "color" en japonés! Fuera de bromas, con el comando COLOR podemos cambiar el color del texto, del fondo y del borde de la pantalla. La sintaxis es:

```
COLOR primer termino,fondo,borde
```

El color de primer término es el mismo del texto; el de fondo es el del total de la pantalla, y el del borde es el de los límites de la misma, donde no se puede escribir. El comando

```
COLOR 15,1,1
```

escribirá letras blancas sobre un fondo negro, siendo el marco de la pantalla también negro. Los colores disponibles y los números que los representan en la sentencia COLOR son los siguientes:

0. Transparente
1. Negro
2. Verde medio
3. Verde claro
4. Azul oscuro
5. Azul claro
6. Rojo oscuro
7. Cián
8. Rojo medio
9. Rojo claro
10. Amarillo oscuro
11. Amarillo claro
12. Verde oscuro
13. Magenta
14. Gris
15. Blanco.

Sólo necesitamos una palabra de explicación acerca del color transparente. Simplemente permite ver a través de él todo lo que hay debajo, es decir, el fondo. Debido a que sólo podemos disponer de dos colores en la pantalla en modo de texto, si cambiamos el del primer término cambiaremos el de todo el texto ya escrito.

Texto en modo gráfico

Pruebe el programa siguiente:

```
10 SCREEN 2
20 PRINT "Hola"
30 GOTO 20
```

No aparece nada en la pantalla, puesto que hay que emplear técnicas especiales para escribir texto en pantallas gráficas. De todos modos, una vez dominadas, dan un alto grado de control sobre la posición del texto en la pantalla y nos permiten mezclar en ella texto y gráficos de alta y baja resolución.

Conseguimos acceso a las pantallas gráficas usando un comando OPEN y un descriptor de periférico. Este nuevo descriptor es llamado GRP:. Igual que éramos capaces de escribir en pantallas de texto utilizando OPEN "CRT:", usamos OPEN "GRP:" para poder hacerlo en pantallas gráficas. La sintaxis completa del comando necesario para poder escribir texto en pantallas gráficas es

```
100 OPEN "GRP:" FOR OUTPUT AS #numero
```

o

```
200 OPEN "GRP:" AS #numero
```

El comando sólo puede usarse correctamente dentro de un programa después de seleccionar un modo gráfico. El parámetro número es el equivalente al número de fichero usado cuando escribíamos texto en ficheros de cinta. De manera similar podemos usar PRINT # y PRINT # USING para escribir texto en las pantallas gráficas. Haga un RESET en su máquina y pruebe el siguiente programa de demostración.

```
10 SCREEN 2
20 OPEN "GRP:" AS #1
30 PRINT #1,"Hola"
40 CLOSE #1
50 GOTO 50
60 REM evita el retorno a Modo Directo
```

Verá aparecer la palabra "Hola" en la esquina superior izquierda de la pantalla. Si interrumpe el programa y vuelve a ejecutarlo, observara que el texto se ha movido hacia abajo una línea. Esto es debido a que el ordenador "recuerda" que, en su ejecución previa, se escribió un *return* al final de la palabra "Hola". Por tanto, escribiremos el nuevo texto de acuerdo con esta operación. De todas maneras, necesitamos un método para posicionar el texto donde queramos en las pantallas gráficas. LOCATE y WIDTH no tendrán efecto hasta que volvamos al modo de texto, cuando sea determinada la anchura especificada en el comando WIDTH.

El comando utilizado para posicionar el texto se llama PRESET. La sintaxis del comando, cuando lo usamos en pantallas gráficas, es

```
PRESET (X,Y)
```

donde X es la posición horizontal del carácter e Y es la vertical. PRESET trabaja en una rejilla de 256 por 92 en modo 2 y de 64 por 48 en modo 3. El pro-

grama siguiente muestra en acción el comando PRESET, utilizado para posicionar aleatoriamente el texto en la pantalla:

```
10 SCREEN 2
20 OPEN "GRP:" AS #1
30 PRESET (RND(1)*256,RND(1)*192)
40 PRINT #1,"Hola"
50 GOTO 30
60 CLOSE #1
```

La sentencia CLOSE es aquí bastante superflua, ya que nunca se ejecuta debido al GOTO de la línea 50. Las coordenadas pasadas al comando PRESET se refieren a la esquina superior izquierda del primer carácter de la cadena a imprimir.

El sistema de coordenadas usado repite la posición 0,0 en la esquina superior izquierda de la pantalla. Debido a la diferencia en el tamaño de los *pixels*, el texto que se imprime en modo 3 es más grande que el de cualquier otro modo. Es excelente para títulos de páginas, así como para muchas otras aplicaciones de este tipo donde se necesiten caracteres de gran tamaño. Para ver en acción este texto grande, limítese a cambiar, en el anterior programa, SCREEN 2 por SCREEN 3. Variar el color del texto impreso en las pantallas gráficas es muy sencillo: sólo tenemos que usar el comando COLOR. Desde luego, ahora tenemos disponibles los dieciséis colores de los modos gráficos. Esto y el grado de libertad que tenemos posicionando el texto confieren una gran utilidad al empleo de texto en modo gráfico. El programa siguiente muestra cómo podemos variar el color del texto con el comando COLOR. El color de las palabras impresas en la pantalla cambiará repetidamente según evoluciona el programa:

```
10 SCREEN 3
20 OPEN "GRP:" FOR OUTPUT AS #1
30 PRESET (0,0)
40 COLOR RND(1)*15
50 PRINT #1,"Hola"
60 GOTO 40
70 CLOSE #1
```

También pueden usarse en estos comandos los parámetros de color de fondo y de borde, aunque en modo gráfico (2 y 3) sólo se puede actuar sobre el color del texto y del borde. El color de fondo usado cuando se imprime texto en las pantallas gráficas es el mismo que estaba utilizándose con el último modo de texto. Si desea cambiar el color de fondo de la pantalla mediante el comando COLOR mientras está en modo gráfico, use la combinación de órdenes siguiente:

```
1010 COLOR primer termino,fondo,borde
1020 CLS
```

Esto, desde luego, borrará la pantalla.

Si, mientras realizamos todos estos cambios de color, accidentalmente regresamos a modo de texto con una combinación de colores completamente ilegible, pulse F6 para poner las cosas como estaban al principio.

Cuando haya escrito en la pantalla gráfica todo el texto que deseaba, debe ejecutar una instrucción CLOSE #1. Si ha usado cualquier otro número de fichero que no sea 1, limítese a sustituirlo por el número utilizado.

Veamos ahora los comandos gráficos más simples disponibles en modos 2 y 3. Lo más sencillo de hacer en términos gráficos es colocar un *pixel* en una determinada posición de pantalla con un color determinado. El sistema de coordenadas utilizado por los comandos gráficos empieza, como siempre, en la esquina superior izquierda de la pantalla. Cuando el intérprete BASIC encuentra coordenadas en comandos, les permite tomar valores más allá de los límites de la pantalla, aunque deben estar en el intervalo −32768, +32767. Los valores que se mantuvieran fuera de la pantalla se sustituirían por su valor más cercano dentro de ella. Así, 0 reemplazaría a cualquier valor negativo especificado como coordenada, si se trata de una referencia absoluta. Se dice que la referencia es absoluta si damos las coordenadas del *pixel* a modificar respecto de 0,0. Es posible especificar la posición de un *pixel* respecto de cualquier otro punto distinto del 0,0. Se dice entonces que se trata de una referencia relativa a algún punto de la pantalla.

Los comandos que podemos usar para dar un determinado color a un *pixel* son PSET y PRESET. Estos dos comandos trabajan de manera idéntica. La sintaxis del comando PSET es:

```
PSET (X,Y),color
```

X e Y especifican las coordenadas del punto, y el parámetro de color, el color que ha de tomar ese punto. El parámetro color puede omitirse en el comando PRESET, que tiene la misma sintaxis. Si se omite el color se tomará el del fondo en ese momento. De esta manera utilizamos PRESET para colocar texto en la pantalla gráfica. Si queremos, podemos usar PSET para hacer la misma función, empleando para el *pixel* el color transparente:

```
PSET (X,Y),0
```

El tamaño del *pixel* depende obviamente del modo gráfico usado. Para comprobarlo teclee el siguiente programa, y efectúelo en ambos modos, es decir, el 2 y el 3:

```
10 SCREEN 2:REM o SCREEN 3
20 PSET (RND(1)*50,RND(1)*50),RND(1)*15
30 GOTO 20
```

Aunque el área de pantalla afectada por el programa es la misma en ambos casos, los *pixels* del modo 2 sólo son la cuarta parte de los del modo 3. Como demostración más palpable, el siguiente programa dibuja un gráfico, mostrando el seno y el

coseno de varios ángulos en dos colores diferentes. Antes de dibujar el gráfico se calculan los senos y los cosenos, para acelerar el proceso. El ángulo en grados, representado por I%, ha de expresarse en radianes antes de aplicar las funciones seno y coseno. A, entonces, contiene el ángulo en radianes. Este programa también es un buen ejemplo de la baja velocidad de cálculo de funciones trigonométricas de los ordenadores MSX.

```
10 SCREEN 0:LOCATE 10,10:PRINT "Espere"
20 DIM S(360),C(360)
30 FOR I%=0 TO 360 STEP 2:A=I%/57.33
40 S(I%)=SIN(A):C(I%)=COS(A)
50 NEXT I%
60 SCREEN 2:REM o SCREEN 3
70 FOR I%=0 TO 360 STEP 2
80 PRESET (I%/2,S(I%)*30+50),7
90 PRESET (I%/2,C(I%)*30+50),1
100 NEXT I%
110 GOTO 110
```

Los gráficos pueden dibujarse tanto en modo 2 como en modo 3, simplemente cambiando adecuadamente la línea 60. La siguiente demostración es lo que se suele llamar en los círculos informáticos un "camino aleatorio". El siguiente *pixel* a trazar se especifica según un cierto grado de factores aleatorios. En este caso particular, las coordenadas X e Y se incrementan o decrementan dependiendo de los valores de dos números aleatorios.

```
10 SCREEN 2
20 X%=100:Y%=100
30 PSET (X%,Y%),15
40 N%=RND(1)*2:M%=RND(1)*2
50 IF N%>0 THEN X%=X%+1 ELSE X%=X%-1
60 IF M%>0 THEN Y%=Y%+1 ELSE Y%=Y%-1
70 GOTO 30
```

Intente alterar las cantidades en que varían X% e Y% en las líneas 50 y 60.

Ahora que podemos usar los comandos PSET y PRESET para trazar *pixels* individualmente, sería interesante ser capaz de dibujar líneas que unan dos puntos de la pantalla. El comando de BASIC MSX que usamos para hacerlo se llama LINE. La sintaxis completa del comando es:

```
LINE (X,Y)-(X1,Y1)
```

X e Y son las coordenadas de la posición donde va a comenzar la línea, y X1, Y1, las de la posición donde termina. La línea dibujada por el comando

```
LINE (10,10)-(100,100)
```

empezará en la posición 10,10 y terminará en 100,100. Podemos especificar el color de una línea de este tipo, ya sea con el comando COLOR antes de dibujarla o por un parámetro de color al final de la instrucción LINE.

```
LINE (10,10)-(30,100),1
```

Este comando dibujaría una línea recta negra entre las coordenadas especificadas. Los números usados en el parámetro de color son los que ya hemos visto. Hay un parámetro final que podemos añadir a un comando LINE, y que es bastante útil. Para dibujar un rectángulo, o "caja", en la pantalla necesitaríamos normalmente cuatro sentencias LINE. La orden

```
LINE (10,10)-(100,100),1,B
```

dibujará una caja negra en la pantalla, cuya esquina superior izquierda estará en la posición 10,10 y la inferior derecha en la 100,100. Podemos reemplazar la letra B por BF. Esta operación dibujará la caja y le dará el color especificado en el comando LINE. Obviamente, las formas dibujadas por este comando son todas rectángulos con lados paralelos. Si quisiéramos dibujar formas irregulares o triángulos, tendríamos que utilizar comandos LINE separados.

Antes de acabar con este comando, veamos cómo podemos trazar puntos o líneas en coordenadas relativas a la posición actual del cursor gráfico, en vez de en coordenadas absolutas. El par de coordenadas X e Y que nos hemos encontrado hasta ahora pueden ser sustituidas por una expresión de la forma

```
STEP (X,Y)
```

STEP significa que las coordenadas están dadas en una referencia relativa. Así, el par de comandos

```
200 PRESET (100,100)
210 LINE STEP (10,10)-(30,60)
```

dibujará una línea desde el punto 110,110 hasta el 30,60. El 10,10 en la expresión STEP se refiere al punto 10,10 respecto al último punto gráfico usado, que en este caso era el punto 100,100 especificado en el comando PRESET.

La forma final de la instrucción LINE es omitir las coordenadas de comienzo. El comando es

```
LINE -(X1,Y1)
```

y la línea se dibuja desde el último punto gráfico trazado hasta el X1, Y1.

Hemos visto cómo podemos dibujar cajas en la pantalla. ¿Y trazar alguna figura más difícil, como un círculo? La mayoría de los ordenadores personales necesitan

que el usuario escriba una subrutina para dibujar círculos; no sucede así con el sistema MSX. El comando CIRCLE nos facilita la posibilidad de dibujar círculos o elipses en cualquiera de los modos gráficos. La sintaxix completa del comando es

```
CIRCLE (X,Y),radio,color,angulo inicial,angulo
              final,razon axial
```

En la expresión anterior puede utilizarse STEP (X,Y) en lugar de (X,Y). Los parámetros color, ángulo inicial, ángulo final y razón axial son opcionales. Empecemos dibujando unos cuantos círculos.

```
10 SCREEN 2
20 X=RND(1)*256:Y=RND(1)*192
30 REM posicion aleatoria del circulo
40 R=RND(1)*30:REM radio aleatorio
50 CIRCLE (X,Y),R,RND(1)*15:REM dibujo
60 REM del circulo con color aleatorio
70 GOTO 20
```

Este programa también funcionará en modo 3, pero los círculos serán trazados con líneas mucho más gruesas. Si va a usar el método STEP(X,Y) para posicionar el círculo, el primero dibujado en el programa anterior dependería de la posición alcanzada por el último comando PSET, LINE, etc. La sentencia CIRCLE puede usarse solamente en los modos gráficos.

Veamos ahora los tres parámetros finales que podemos utilizar con el comando CIRCLE; los ángulos inicial y final, y la razón axial. Se usan cuando dibujamos sectores de círculos o elipses. Empecemos viendo cómo podemos dibujar arcos de circunferencia en la pantalla.

Hacemos esto especificando los ángulos inicial y final en el comando CIRCLE. Estos parámetros han de estar en radianes, pero podemos servirnos de la función diseñada hace un momento, que convierte ángulos en grados a radianes, para resolver este problema. Yo siempre utilizo los grados para la medida de ángulos, porque creo que la mayoría de la gente está más familiarizada con estas unidades. Los ángulos usados en estos dos parámetros deben estar entre 0 y 360 grados. Los ordenadores MSX dibujan círculos como el de la figura 6.1.

El círculo se dibuja en la dirección contraria a las agujas del reloj, empezando en el ángulo inicial y terminando en el ángulo final, como cabía esperar. Así, cuando dibujamos con un comando CIRCLE si no se especifican estos parámetros, el sistema asume para ellos 0 y 360 grados respectivamente. El programa siguiente le permite ver los efectos que produce el cambiar el ángulo inicial en el proceso de trazar círculos. La línea 20 del programa define la función de nuestra conversión de grados a radianes. En la línea 30 llamamos a la función, colocando el ángulo inicial pedido como argumento de la misma. De esta manera puede cambiar el ángulo inicial alterando el parámetro pasado a la función.

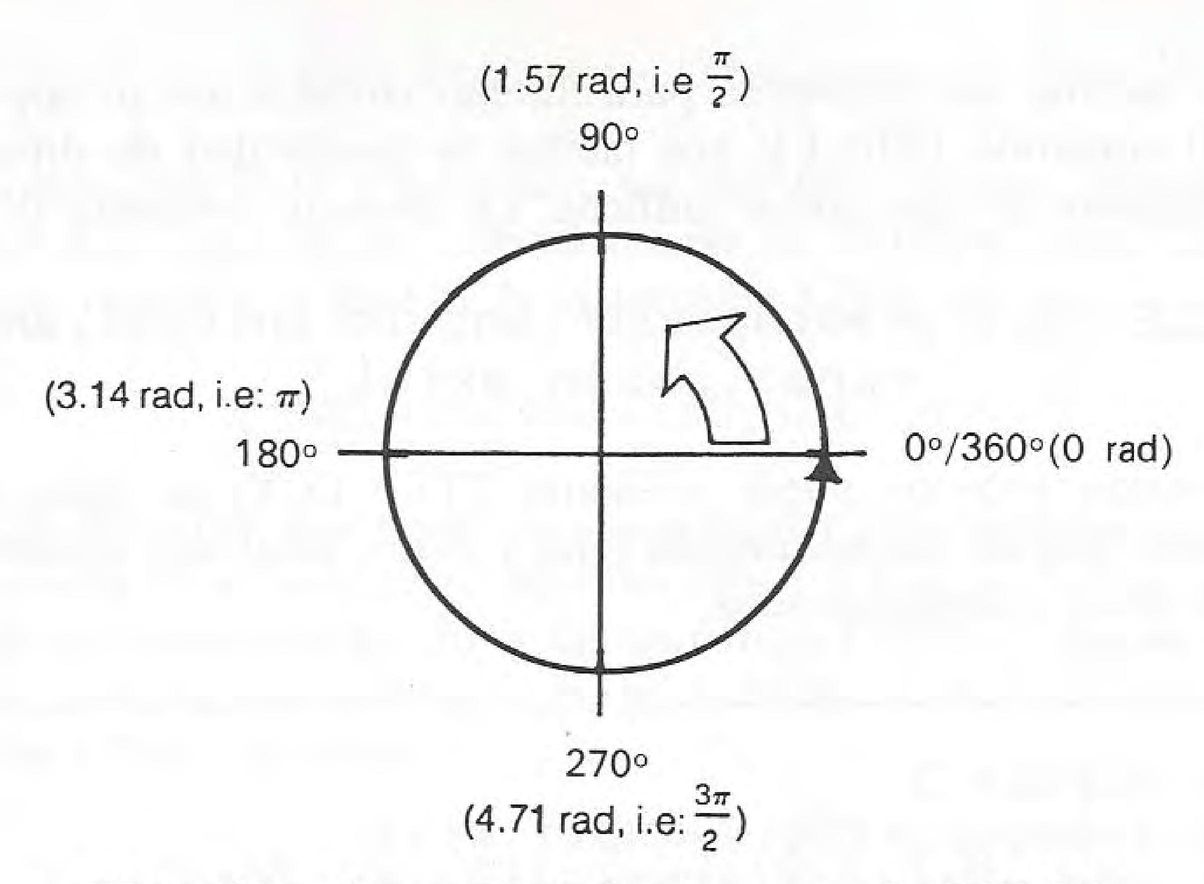


Figura 6.1.—*Convención de ángulos, sentencia CIRCLE.*

```
10 SCREEN 2
20 DEF FNA(ANGULO)=ANGULO/57.33
30 CO=FNA(180):REM cambie el argumento
40 REM de esta funcion si quiere variar
45 REM el angulo inicial
50 CIRCLE (100,100),40,15,CO
60 GOTO 60
```

La figura 6.2 muestra los efectos de variar el argumento.

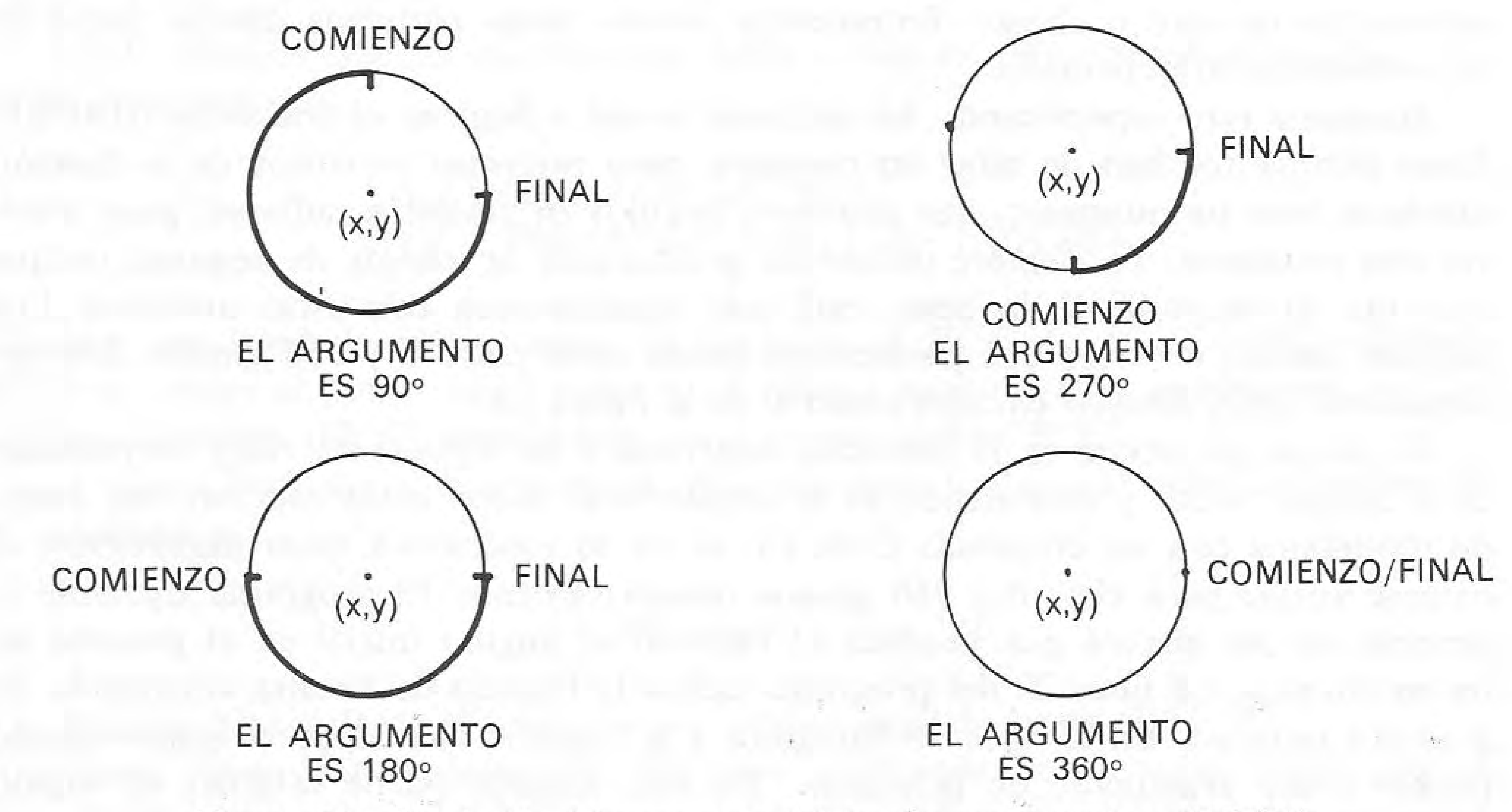


Figura 6.2.—*Variación del argumento del ángulo suministrado a CIRCLE.*

Es posible dibujar arcos de circunferencia especificando el parámetro de ángulo final en el comando CIRCLE, como hemos hecho con el ángulo inicial. Así, si dibujásemos un círculo con el ángulo inicial igual a 0 grados y de ángulo final igual a 90 grados tendríamos una línea describiendo un cuarto de circunferencia. La longitud del arco producido depende del radio del círculo, y la posición del arco se determina por las coordenadas X,Y especificadas en el comando CIRCLE. De nuevo experimente variando los valores del parámetro de ángulo final, recordando que deben darse los ángulos en radianes a la sentencia CIRCLE.

El parámetro final del comando CIRCLE, la razón axial, se usa cuando queremos dibujar elipses. Como seguramente sabrá, una elipse es un círculo "aplastado". La razón axial de la elipse mide la proporción en que está achatada en relación con un círculo. Es la relación entre el radio vertical y el horizontal de la figura. En un círculo, estos dos radios son iguales, y por eso tiene la razón axial de 1. La figura 6.3 muestra dos elipses con sus respectivas razones axiales. Si éstas son muy grandes, conducirían a una línea recta, al igual que si fueran muy pequeñas. El centro de la elipse se especifica de nuevo por las coordenadas X,Y del comando CIRCLE.

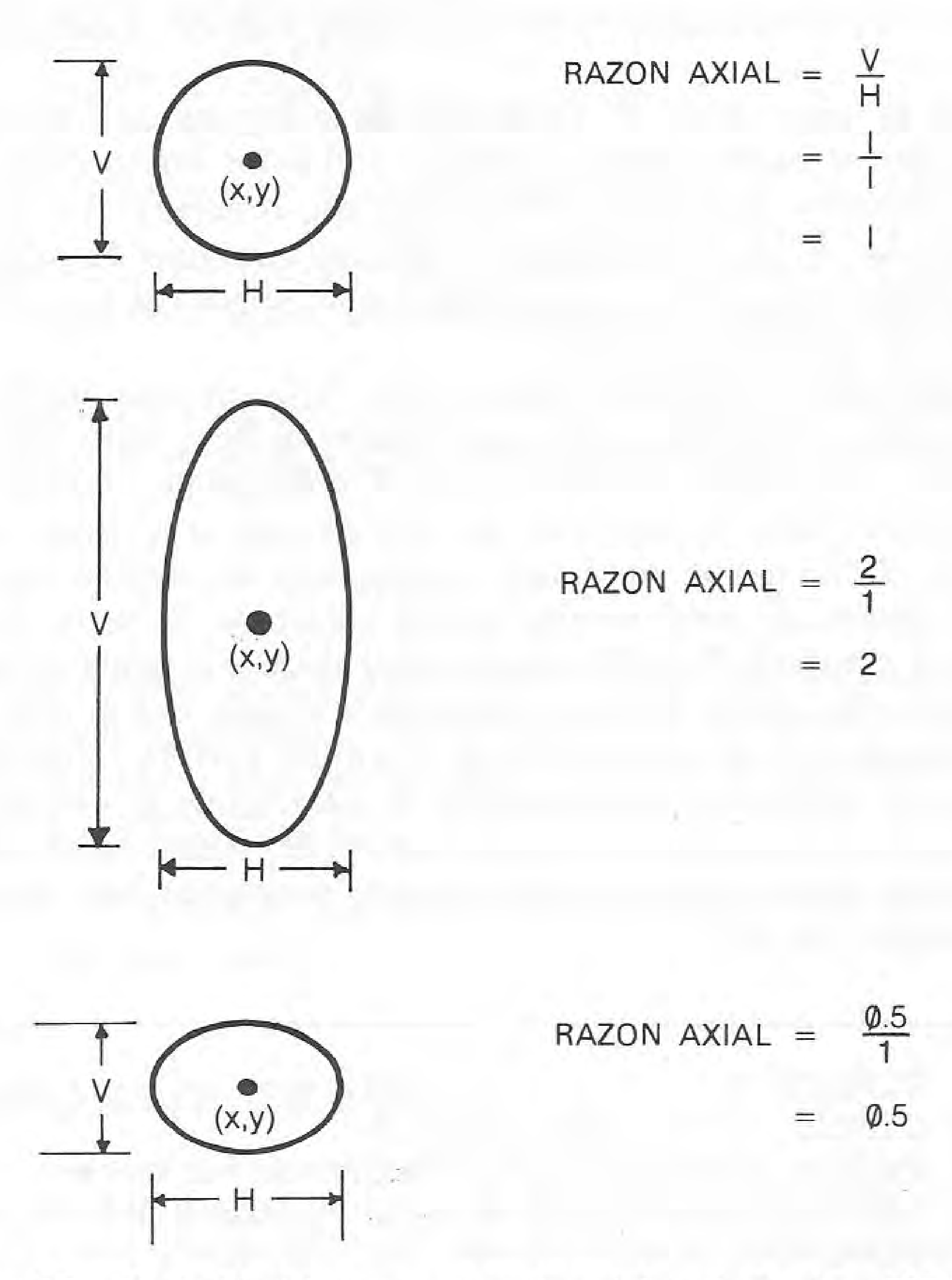


Figura 6.3.—*Variación del argumento razón axial suministrado a CIRCLE.*

Un punto interesante para todos los usuarios del comando CIRCLE es que, si los ángulos especificados son negativos, éstos serán tratados como positivos, y el perímetro de la circunferencia o elipse y el punto X,Y se unirán por sendas líneas rectas.

PAINT

Sabemos cómo podemos dibujar rectas o círculos de colores especificando el parámetro de color en la instrucción. Podemos trazar *pixels* de colores de una manera similar. Sin embargo, ¿qué ocurre si queremos rellenar un círculo o alguna figura irregular de color? Una posibilidad sería trazar cada *pixel* dentro de la figura del color requerido. Esto funcionaría, pero en BASIC llevaría mucho tiempo lograrlo. El BASIC MSX nos proporciona una rutina en ROM basada en este principio pero en código máquina, lo que significa que es mucho más rápida que el BASIC. Esta rutina se llama PAINT, y el comando tiene la sintaxis siguiente:

```
PAINT (X,Y),color,color a considerar como borde
```

Puede usarse la forma STEP (X,Y) en lugar de la expresión (X,Y). Ambos colores, el color con el que se desea rellenar la figura, y el color límite, como explicaremos pronto, son parámetros opcionales. ¿Cómo usar este comando? La coordenada X,Y especifica la posición donde el ordenador empezará a rellenar de color la figura. Lo hace sobre una base línea a línea, y reconoce los extremos del área a llenar. Operación que realiza de la siguiente manera:

Como la operación PAINT se efectúa a lo largo de una recta, el ordenador comprueba el color de todos los *pixels* que encuentra. Si el color del *pixel* no es el especificado como color límite, se cambia por el definido para rellenar la figura. Si el *pixel* es del color límite, la operación no va más lejos a lo largo de esa recta en ese sentido. Así, si dibujamos un círculo y colocamos las coordenadas X,Y del comando PAINT dentro de dicho círculo, éste se coloreará. Si el par de coordenadas especificado en el comando PAINT está fuera del círculo, se pintará toda la pantalla excepto el interior del mismo. En estos ejemplos se supone que el color de la circunferencia es el mismo que el color límite de la rutina PAINT. Una advertencia interesante es que el parámetro que especifica el color límite se descarta en el modo de pantalla 2. Aquí, el color límite es el mismo que el de relleno. En el modo 3 se puede especificar un color límite. Veamos ahora algunos programas que demuestran el potencial del comando PAINT.

```
10 SCREEN 3
20 CIRCLE (100,100),40,15
30 PAINT (100,100),5,15:REM diferentes
40 REM colores para pintar y para el borde
50 GOTO 50
```

Observe que hemos fijado el color límite del mismo que dibujamos el círculo. Ejecute este programa y añada la línea siguiente, que rellenará el fondo con diferente color.

```
45 PAINT (100,190),1,15
```

De nuevo fijamos el color límite con el color de círculo. Debido al hecho de que este comando especifica directamente el color límite, no funcionará correctamente en el modo de pantalla 2. A continuación se expone una demostración del uso de PAINT en modo 2:

```
10 SCREEN 2
20 CIRCLE (100,100),40,5
30 CIRCLE (100,100),50,15
40 CIRCLE (100,100),60,15
50 CIRCLE (100,100),70,1
60 PAINT (100,100),5
70 PAINT (100,152),15
80 PAINT (100,190),1
90 GOTO 90
```

Está claro que se ha de tener más cuidado usando el comando PAINT en modo 2; el color del contorno de la figura en modo 2 debe ser el mismo con que intentamos rellenarla.

El comando gráfico final que vamos a ver se llama POINT. Esta instrucción nos permite conocer el color de un *pixel* en una determinada posición de pantalla. La sintaxis es

```
POINT (X,Y)
```

y la sentencia que contiene el comando POINT asigna generalmente a una variable el valor obtenido por el comando. Este retorna un valor entre 0 y 15. X e Y son las coordenadas que dan la posición del punto. Un ejemplo típico del uso de POINT lo podemos ver en la sentencia:

```
200 IF POINT(x,y)=1 THEN GOSUB 2000 ELSE
    GOSUB 3000
```

Macrolenguaje gráfico[2]

Este título tan largo se da a una serie de comandos gráficos basados en la orden DRAW. La instrucción DRAW y sus cadenas de funciones asociadas forman un lenguaje dentro del BASIC. La sintaxis del comando DRAW es

DRAW cadena de comandos graficos

Este comando sólo trabaja en los dos modos gráficos, y cualquier orden pasada a la instrucción DRAW en la cadena de comandos gráficos comenzará a dibujar desde el último punto referenciado; es decir, el último punto dibujado o trazado por un comando gráfico de cualquier tipo.

La cadena de comandos gráficos consiste en determinadas letras y números, así como de algunos caracteres no alfanuméricos. Los comandos más simples del macrolenguaje gráfico son aquellos que se usan para dibujar líneas rectas. En cada comando listado a continuación, *n* es el número de *pixels* que se quiere dibujar.

Comando	**Función**
U n	Dibuja hacia arriba
D n	Dibuja hacia abajo
L n	Dibuja hacia la izquierda
R n	Dibuja hacia la derecha
E n	Dibuja diagonalmente, arriba y a la derecha
F n	Dibuja diagonalmente, abajo y a la derecha
G n	Dibuja diagonalmente, abajo y a la izquierda
H n	Dibuja diagonalmente, arriba y a la izquierda.

El programa de demostración siguiente dibuja un cuadrado. Observe que empleamos el comando PRESET para colocar el cuadrado y definir su color.

```
10 SCREEN 2
20 PRESET (100,100),15
30 DRAW "U50L50D50R50"
40 GOTO 40
```

Si quisiéramos dibujar líneas con un ángulo particular, podemos usar el comando M X, Y, que es similar en muchos aspectos a la sentencia LINE que ya hemos estudiado. Si deseáramos utilizar el comando M para dibujar desde el último punto referenciado, distinto de 0, a un punto relativo, de igual manera que usabamos la palabra STEP en LINE procederemos ahora a anteponer un signo + a – a las coordenadas X o Y. Por ejemplo:

```
1000 DRAW "M+10,100"
```

trazaría una línea hasta un punto situado 10 *pixels* a la derecha, y 100 hacia abajo desde la posición gráfica actual. El signo negativo nos permite referenciar un punto a la izquierda de X y por "encima" de Y en sus posiciones actuales. Recuerde siempre que el comando M dibuja una línea en el color —también actual— de primer término. Si desea trasladarse a un punto de la pantalla sin dibujar la línea, puede usarse cualquiera de los comandos vistos hasta ahora, precedidos de la letra "B".

El comando "N" le permite dibujar una línea con cualquiera de los comandos previamente examinados, y regresar a la posición inicial donde estaba antes de ejecutarse la sentencia precedida de una N. Así, el comando

```
DRAW "M100,100NU50"
```

dibujará primero una línea hasta la posición 100,100 y después otra hasta la posición 100,50. Sin embargo, la siguiente línea a dibujar comenzará en la posición 100,100, que es la de comienzo del comando precedido por N. Para exponerlo con más claridad, teclee el programa siguiente y ejecútelo:

```
10 SCREEN 2
20 PRESET (100,100),1
30 DRAW "NU100NL100NL100NR100ND100"
40 GOTO 40
```

La N situada antes de cada comando de la cadena provoca que cada uno de ellos comience a dibujar desde la misma posición.

Color

Es fácil realizar cambios de color en el macrolenguaje gráfico. Esto se consigue insertando el carácter "C" en la cadena de comandos, seguido de un número entre 0 y 15, que representa el color deseado. Los números del código de colores son aquellos que hemos tratado previamente para su uso en los modos de texto y gráficos. Así, la línea siguiente dibujará en la pantalla un cuadrado negro:

```
DRAW "C1U50L50D50R50"
```

Las sentencias de color pueden colocarse en cualquier lugar de la cadena de comandos de DRAW.

Otra orden que se usa es la de ángulo, "A". Un inconveniente de este comando es que no hay un gran número de ángulos disponibles (de hecho, sólo son cuatro...). La sintaxis dentro de la cadena de comandos de DRAW es An, donde *n* tiene un valor de acuerdo con la tabla siguiente. Como en la instrucción CIRCLE, los ángulos se miden en el sentido contrario a las agujas del reloj.

Valor de *n*	**Angulo**
0	0 grados
1	90 grados
2	180 grados
3	270 grados

La siguiente demostración dibujará un cuadrado usando los comandos ángulo en lugar de las órdenes D, L y R:

```
10 SCREEN 3
20 PRESET (100,100),1
30 DRAW "A1U90A2U90A3U90A0U90"
40 GOTO 40
```

Factor de escala

En circunstancias normales, el parámetro *n* pasado a comandos como U o D corresponde directamente al número de *pixels* a trazar. Un comando de la forma Sn, donde *n* es un valor entre 0 y 255, nos permite variar el número de *pixels* representados por un determinado valor de *n* en estas sentencias. El valor de $n = 4$ es el factor de referencia de escala, y por eso si tenemos un valor de $n = 1$ la figura se dibujará a 1/4 de su tamaño normal. Similarmente, si establecemos $n = 8$, emitiendo la orden S8, la figura dibujada será del doble de tamaño del que tendría sin factor de escala.

Subrutinas

Sería interesante tener, dentro del macro lenguaje gráfico, el equivalente de la subrutina BASIC; esto es, un medio de mantener sólo una copia de una serie de comandos, pero siendo capaces de ejecutarlos siempre que queramos. Podemos, de hecho, hacerlo usando el comando X y variables de cadena. Por ejemplo, podríamos definir cu$ de tal manera que contenga los comandos que dibujan un cuadrado. En tal caso es factible llamarla dentro de otra cadena como parte de un comando DRAW usando la instrucción

```
Xcu$;
```

La totalidad de las sentencias necesarias para dibujar un cuadrado de esa manera sería:

```
A$="U30R30D30L30"
DRAW "XA$;"
```

El programa siguiente lo muestra con más detalle, e indica cómo una cadena usada en un comando X puede llamar por sí misma a otras cadenas con comandos X:

```
10 SCREEN 2
20 a$="U30"
30 b$="Xa$;L30D30R30"
40 PRESET (100,100),1
50 DRAW "Xb$;"
60 GOTO 60
```

Por tanto, usando la orden X es posible unir cadenas de comandos para la instrucción DRAW que tenga, en efecto, más de 255 caracteres. Cualquier nombre de variable de cadena usado de este modo debe preceder a un punto y coma.

Variable en el macrolenguaje gráfico

En todos los comandos vistos, los parámetros numéricos han sido siempre constantes. Pero no es necesario: podemos usar una variable normal BASIC que haya sido declarada, a lo largo del programa, dentro de la cadena de comandos de DRAW. El nombre de variable usado ha de estar precedido por un signo "=", y seguido por un ";". Así tendríamos:

```
100 arriba=100
110 DRAW "U=arriba;D30R30"
```

Las expresiones están prohibidas en las cadenas pasadas al comando DRAW. Deben evaluarse previamente y asignarse los valores obtenidos a la variable contenida en la cadena.

En todas estas sentencias puede usarse el signo ";" como separador de comandos opcional. Los espacios se ignoran cuando se interpreta la cadena del comando DRAW. Evidentemente, el macrolenguaje gráfico sólo puede usarse en modos gráficos.

Sprites

Hemos visto cómo podemos escribir texto en pantallas gráficas, y cómo dibujar rectas y circunferencias usando los comandos gráficos. Sin embargo, una vez comencemos a escribir *software* de juegos, por ejemplo, empezaremos a necesitar caracteres que parezcan invasores del espacio, pepinos gigantes... ¡o lo que sea, según el contenido de nuestro juego! Podemos crear fácilmente estos caracteres para utilizarlos en modo gráfico. Se les llama *sprites* y son extremadamente útiles en la programación de gráficos. Un *sprite* puede definirse en términos simples como un carácter del que podemos definir su forma y mover por la pantalla. También podemos decir si dos *sprites* colisionan en su recorrido por la pantalla. Pueden ser más grandes que los caracteres normales, y un *sprite* puede estar compuesto de más de un carácter. No importa de

cuántos caracteres esté compuesto; para los propósitos de la programación, el ordenador lo trata como si fuera una unidad en BASIC.

Antes de seguir y examinar cómo podremos utilizar y definir los *sprites,* conviene estudiar cómo trata a los *sprites* el procesador de *display* de video.

La imagen que presenta el VDP en la pantalla del monitor o del televisor puede imaginarse como compuesta de una serie de capas, llamadas PLANOS. La figura 6.4 muestra un diagrama. Las imágenes de los planos de *display* más cercanos al observador parecen pasar por delante de los que están en los más lejanos. Esto confiere a los *sprites* la capacidad de pasar por delante o por detrás de otros. Observe que no es posible que un *sprite* pase por detrás de un objeto que esté en un plano más lejano que del *sprite* en cuestión. Se dice que los planos del *display* que están más cerca del observador tienen PRIORIDAD más alta que los que están más lejos. Por tanto, el plano de *display* donde se ve el *sprite* 0 tiene prioridad más alta que el plano donde se ve normalmente el *sprite* 3. Hay treinta y dos planos de *display;*

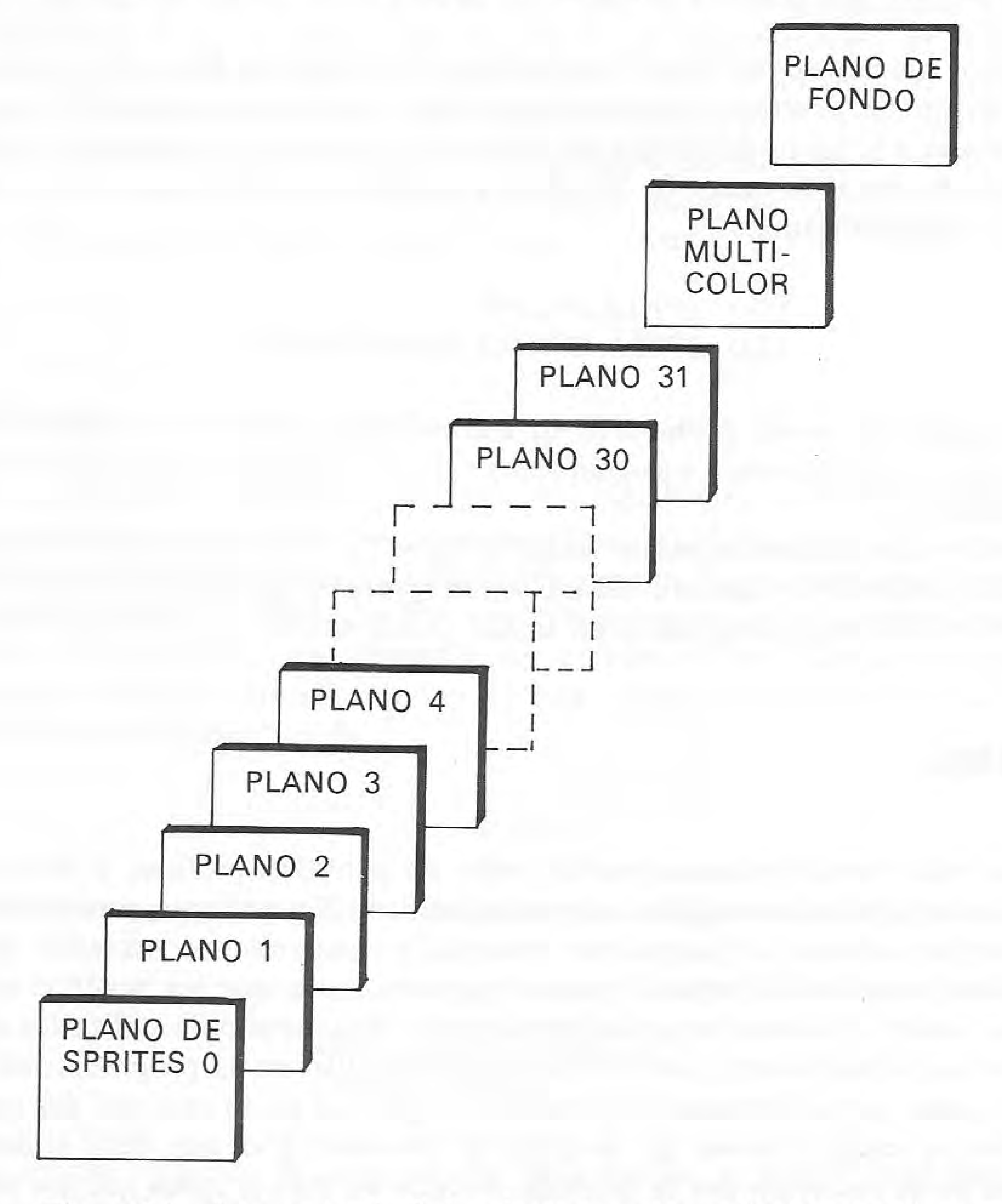


Figura 6.4.—*Representación de los planos de* sprites, *multicolor y de fondo.*

esto explica por qué no es posible, excepto usando técnicas de programación muy avanzadas, tener más de 32 *sprites* en la pantalla a la vez, ya que sólo puede haber un *sprite* por plano. Después de los planos de *sprites* va un plano de *display*, conocido como el "plano multicolor". Es en este plano donde aparecen las imágenes creadas por el manejo de textos y gráficos normales. Así, las imágenes producidas por comandos como PRINT y DRAW tendrán siempre prioridad más baja que las imágenes de *sprites*. El plano de *display* final accesible por el usuario se llama "plano de fondo". Habitualmente vemos esto como el marco o borde de la pantalla.

Los *sprites* no están disponibles en modo 0, pero sí en el resto de ellos, incluyendo el modo de texto 1. Los *sprites* pueden ser de varios tamaños. El tamaño de *sprite* más pequeño es el de 8 por 8 *pixels;* da una imagen del mismo tamaño que un carácter de texto en la pantalla modo 2. El otro tipo es de 16 por 16 *pixels*. Ambos tipos pueden ser "aumentados" en la pantalla, haciéndolos así más grandes.

Por tanto, hay cuatro diferentes tipos de *sprites* disponibles para el programador MSX. El color del *sprite* se define cuando lo posicionamos en la pantalla. Si no se colorea un *pixel* de un *sprite*, podemos ver las imágenes de planos de más baja prioridad a través de los *pixels* transparentes del *sprite*. Por eso, si pasa un *sprite* por encima de otro, el color del *sprite* de prioridad más baja será visible a través de las áreas transparentes del *sprite* de prioridad más alta.

La información necesaria para definir un *sprite* se almacena en la RAM de video en un área de bytes llamada TABLA GENERADORA DE *SPRITES*. (Un examen detallado de esta tabla lo veremos después, en este mismo capítulo.) Como siempre hay una cantidad de memoria dada disponible para la definición de *sprites*, cuanto más grandes son los *sprites* menos podemos definir, ya que un *sprite* de 16 por 16 *pixels* ocupa más memoria que uno de 8 por 8 *pixels*. La memoria necesaria para definir un *sprite* de 8 por 8 *pixels* es la misma si el *sprite* está o no aumentado. Podemos tener hasta 256 modelos de *sprites* distintos para *sprites* de 8 por 8, y hasta 64 para los de 16 por 16.

Tipos de "sprites"

El tipo de los *sprites* a visualizar en la pantalla se define con el comando SCREEN de la siguiente forma:

```
SCREEN modo,tipo
```

donde "modo" es un modo de pantalla capaz de visualizar *sprites*, y "tipo" es un número entre 0 y 3. Los tipos de *sprites* especificados por el parámetro tipo son los siguientes:

Tipo	Tipo de SPRITE
0	8 por 8
1	8 por 8 aumentado
2	16 por 16
3	16 por 16 aumentado

La figura 6.5 muestra el aspecto de los distintos tipos de *sprites* en la pantalla.

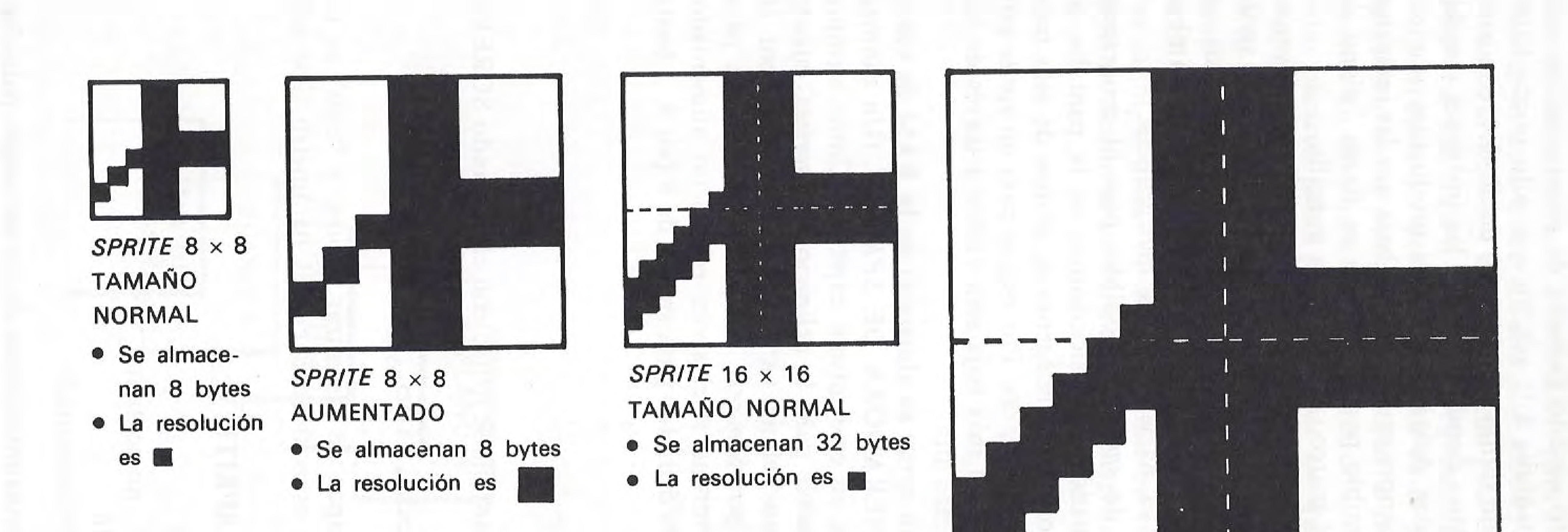


Figura 6.5.—*Definición de* sprites *(1): tipo, memoria y resolución.*

Un *sprite* de 8 por 8 puede explicarse simplemente definiendo un sólo carácter, pero el *sprite* de 16 por 16 requiere que el usuario especifique cuatro secciones de 8 por 8, representando cada sección una cuarta parte del *sprite* completo. Luego usando *sprites* de 16 por 16 *pixels* podremos definir imágenes grandes y moverlas por toda la pantalla como si fueran *sprites* de 8 por 8. Por tanto, ¿por qué no empezamos a definir *sprites*?

Definición de "sprites"

Sin importar de qué tamaño sea el *sprite*, la base de la definición de *sprites* es el uso de una matriz de 8 por 8 cuadrados, como se muestra en la figura 6.6. Observe los números que aparecen a lo largo del margen superior de la matriz: si ha consultado el apéndice, los reconocerá como potencias de 2. Con este instrumento, válido para representar *sprites* de 8 por 8 *pixels*, podemos empezar a definir *sprites*. Para ello, sombreamos cada cuadrado de la matriz que queremos que aparezca del color de primer término cuando se visualice el *sprite*. Es decir, si queremos un *sprite* que represente a un hombrecillo, utilizaríamos una matriz como la sombreada en la figura 6.7.

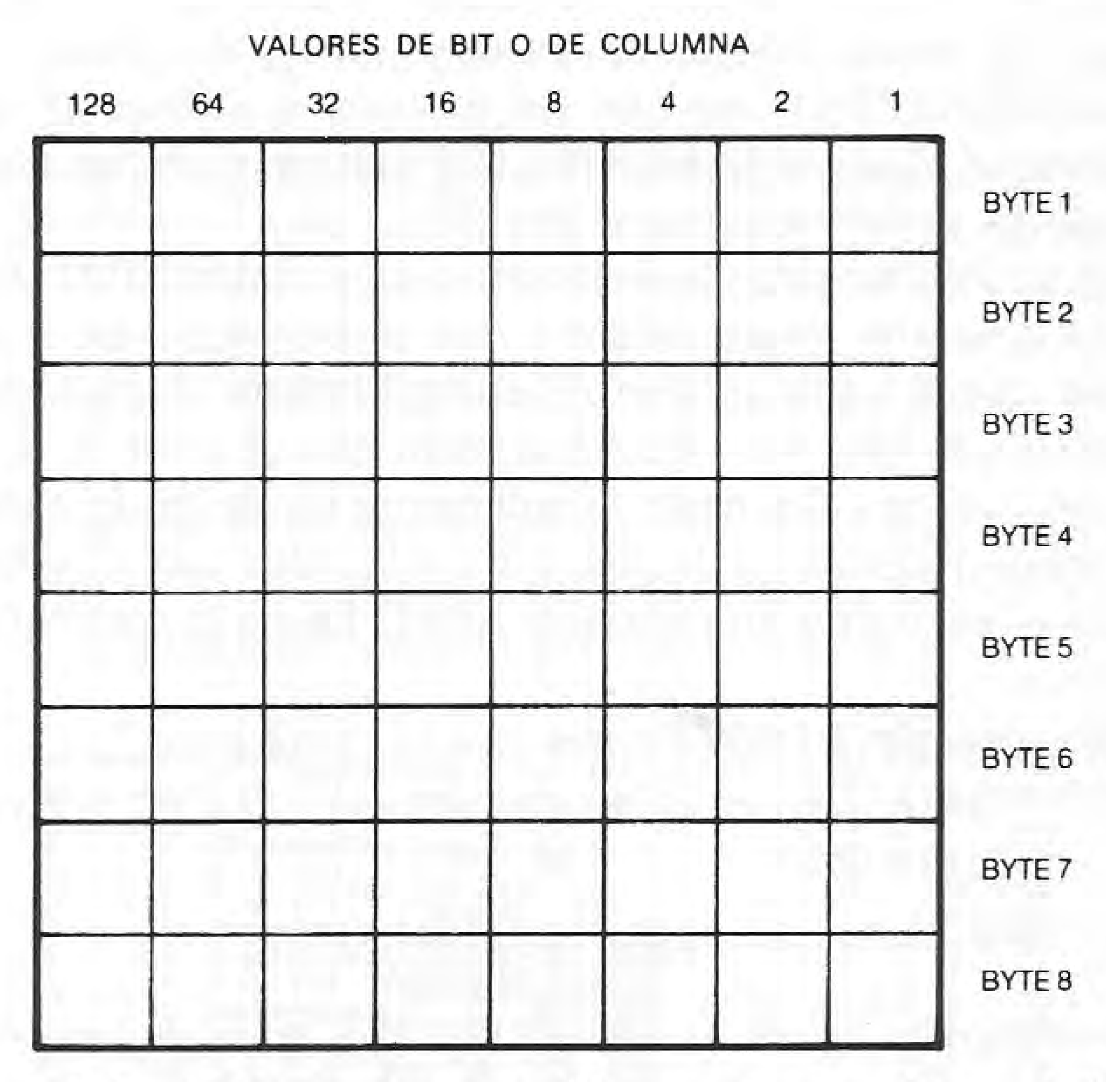


Figura 6.6.—*Definición de* sprites *(2): trazado en un bloque de 8 × 8.*

Para convertirlo en una expresión comprensible por el ordenador utilizamos una serie de variables de cadena especiales llamadas SPRITE$ (*n*). *n* es un entero entre 0 y 255 cuando el tipo de *sprite* es 0 ó 1, y entre 0 y 63 si el tipo de *sprite* es 2 ó 3. Estas variables contienen el patrón del *sprite*, y ocupan 32 bytes en memoria. Para definir un *sprite* de 8 por 8 *pixels* se necesitan sólo 8 bytes, y para definir uno

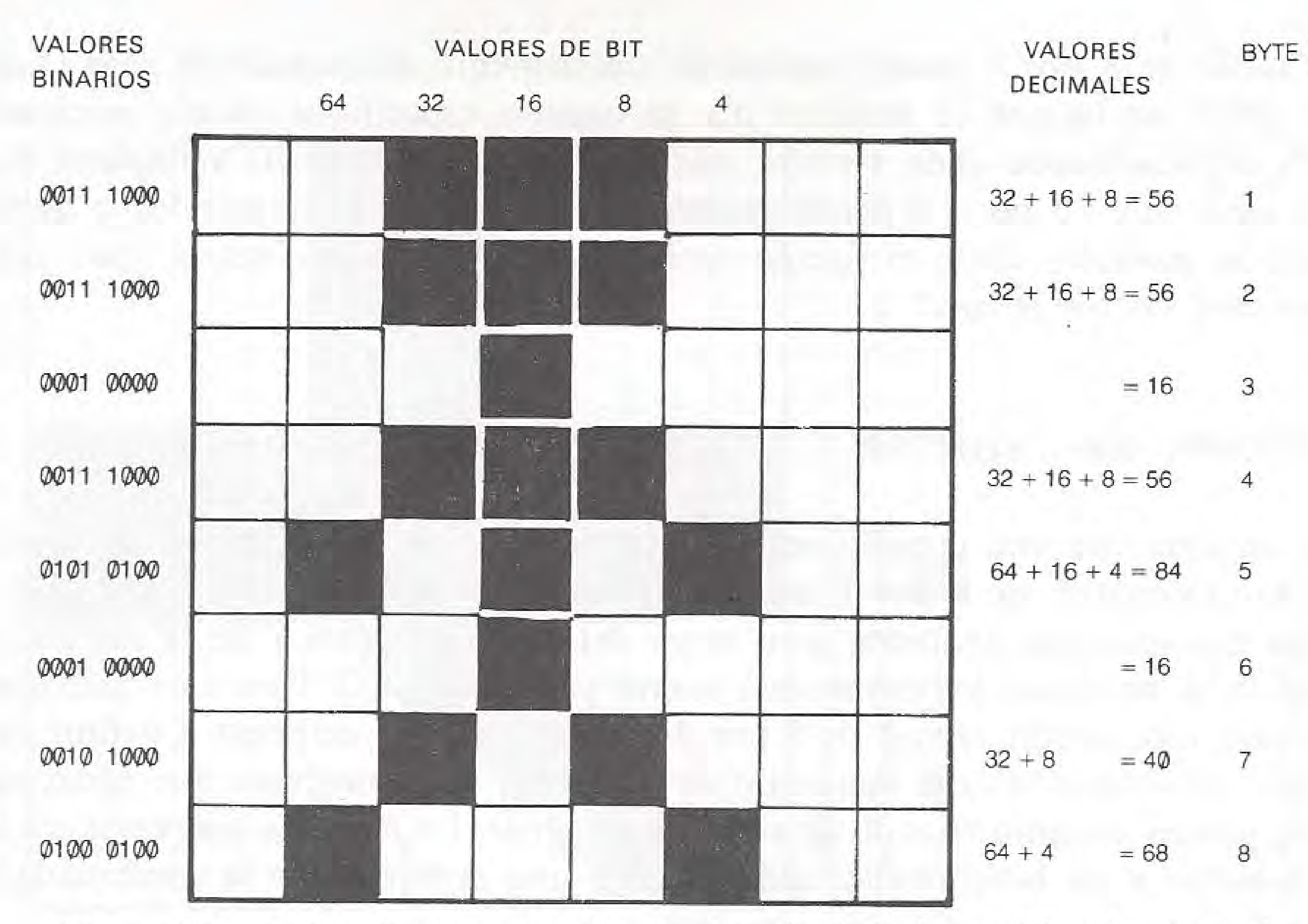


Figura 6.7.—*Definición de* sprites *(3): traducción de la forma del* sprite *de 8 × 8.*

de 16 por 16, que se puede considerar como 4 *sprites* de 8 por 8 *pixels,* son necesarios 32 bytes. SPRITE$(1) contiene los datos que definen el *sprite* visualizado en el plano de *display* 1. Las variables SPRITE$ son para ello un medio de acceder a las tablas generadoras de *sprites* desde el BASIC.

Para transferir la información de la matriz a la variable SPRITE$ podemos convertir cada fila de la matriz en un número. Así se ha hecho en la figura 6.8 con el hombrecillo. Estos valores numéricos se obtienen sumando los valores de las columnas que contienen *pixels* que queremos que aparezcan con el color de primer término en la pantalla. Por eso, en una fila dada, si solamente ha de ser trazado el *pixel* más a la izquierda, el valor asignado a la fila sería 128. Una vez calculados los valores para cada fila, los asignamos a una variable SPRITE$ de la siguiente manera:

```
100 a$=CHR$(56)+CHR$(56)+CHR$(16)
    +CHR$(56)+CHR$(84)+CHR$(16)+CHR$(40)
    +CHR$(68)
110 SPRITE$(1)=a$
```

El contenido numérico de una variable SPRITE$ dada puede ser examinado con la rutina siguiente:

```
10 SCREEN 1
20 n=1
30 FOR I=1 TO LEN(SPRITE$(n))
40 PRINT ASC(MID$(SPRITE$(n),i,1))
50 NEXT I
```

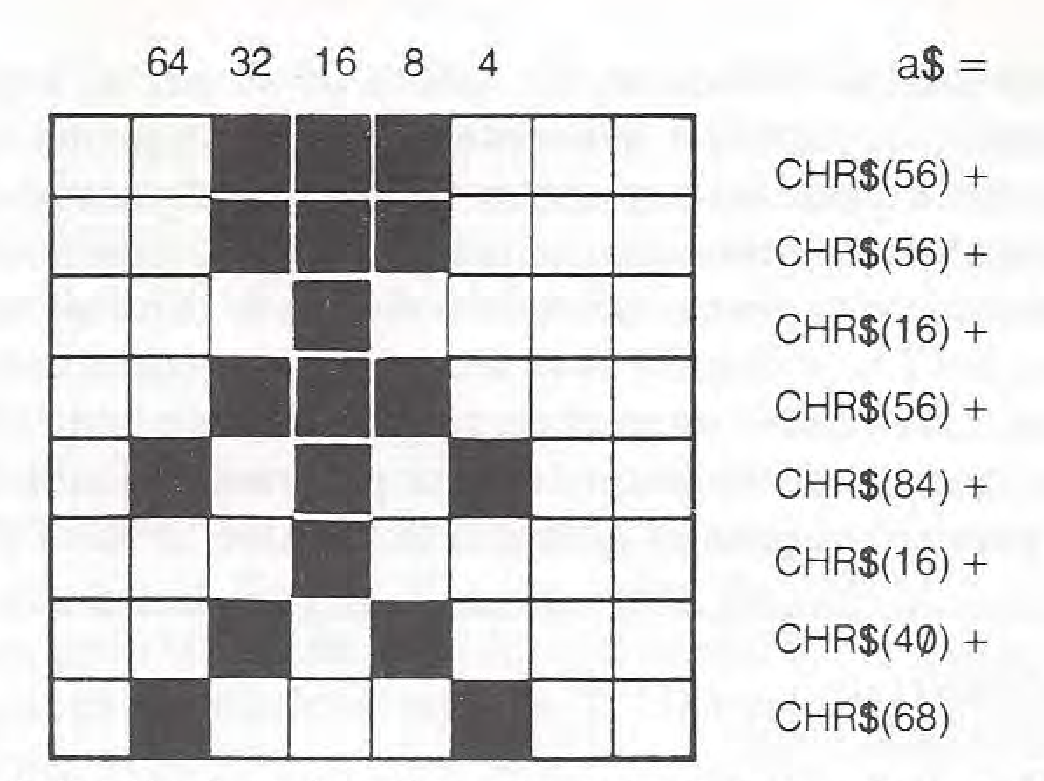


Figura 6.8.—*Definición de* sprites *(4): asignación de datos a la variable SPRITE$.*

Puede asignar una cadena de caracteres a la variable SPRITE$ de la forma habitual, siempre que sea más corta que treinta y dos caracteres. Observe que los contenidos de todas las variables SPRITE$ pueden borrarse mediante una rutina que las fije a una cadena de CHR$(0). El comando SCREEN modo, tipo, también inicializa todas las variables SPRIT$.

Respecto al *sprite* de 16 por 16 *pixels,* los principios de diseño son los mismos. Nos limitamos a diseñar un *sprite* separado de 8 por 8 *pixels* por cada cuarta parte del grande, como en la figura 6.9. Para definir un *sprite* de 16 por 16 *pixels* simplemente diseñamos por orden cada *sprite* de 8 por 8 *pixels,* siguiendo la secuencia numérica de la figura 6.9. Una vez están diseñados todos los componentes del *sprite,* se evalúan los valores numéricos de cada fila y se colocan en la variable SPRITE$ del modo que hemos visto. No obstante, el orden en que se asignan los números a la variable SPRITE$ es importante, y lo puede variar en la línea siguiente:

100 SPRITE$ (1)= cadena del primer cuadrante + cadena del segundo cuadrante + cadena del tercer cuadrante + cadena del cuarto cuadrante

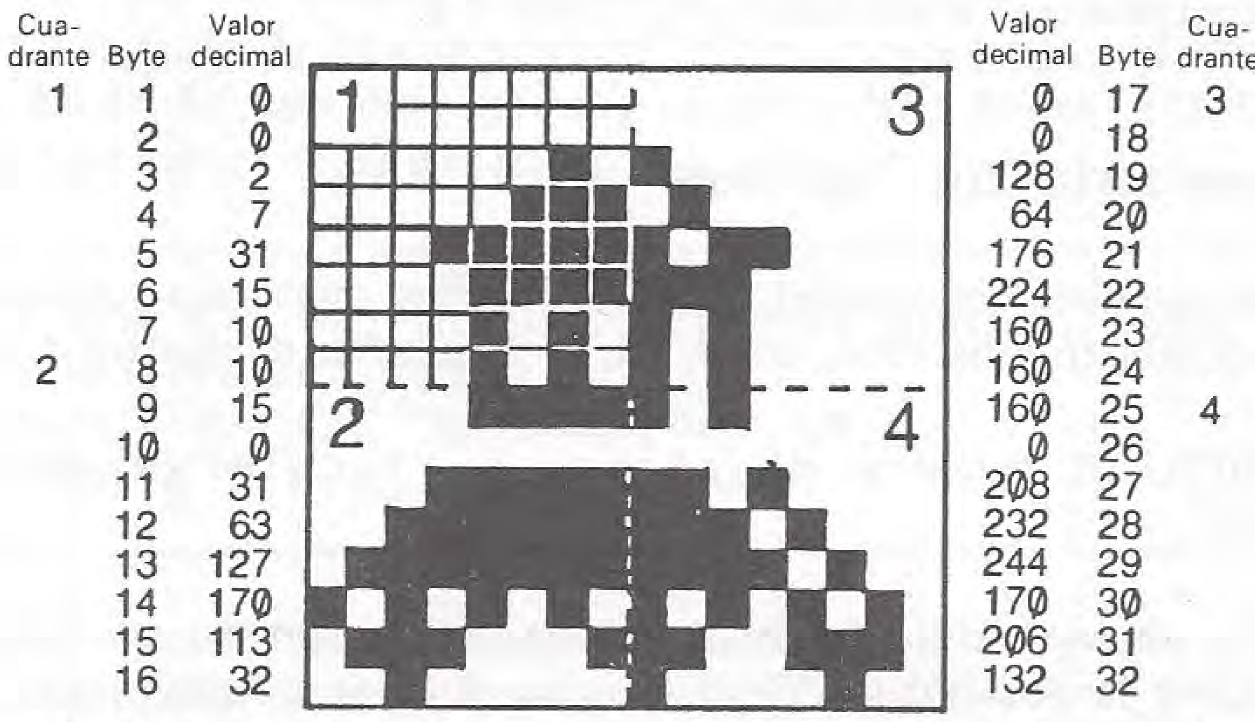


Figura 6.9.—*Definición de* sprites *(5): traducción de la forma de un* sprite *16 × 16.*

Para hacer más fácil la definición de *sprites* de 8 por 8, pruebe el programa siguiente. No es perfecto y puede ser mejorado fácilmente. Las ocho sentencias DATA del final del programa representan las ocho filas de *pixels* de un *sprite* de 8 por 8. Cada "1" de estas sentencias representa un *pixel* del *sprite* coloreado, y cada "0", un *pixel* a dejar en blanco en la forma definitiva. Así, para cambiar la forma del *sprite*, edite las sentencias DATA, y cuando el patrón de estas sentencias sea el que quiere, ejecute el programa. Los valores numéricos que representan cada fila serán visualizados de tal manera que puede incluirlos en sus programas. Estos números están en el orden correcto para su asignación directa a la variable SPRITE$:

```
10 SCREEN 0
20 RESTORE
30 FOR J=1 TO 8
30 READ n$
40 car=0
50 FOR I=8 TO 1 STEP -1
70 X$=MID$(n$,i,1):n=VAL(X$)
80 IF n=1 then car=car+2^(8-I)
90 NEXT
100 PRINT car
110 NEXT
120 END
1000 DATA "10001000"
1010 DATA "01010000"
1020 DATA "00100000"
1030 DATA "00111111"
1040 DATA "00111111"
1050 DATA "00100001"
1060 DATA "00100001"
1070 DATA "00100001"
```

Tras haber definido nuestros *sprites*, el paso siguiente es... ¡usarlos! Para colocarlos en la pantalla empleamos el comando PUT SPRITE.

Posicionamiento de "sprites"

La sintaxis completa del comando PUT SPRITE se muestra a continuación, mientras que los parámetros listados, color, (X,Y) y número de patrón son opcionales.

```
PUT SPRITE numero de plano,(X,Y),color,numero de
patron
```

Si se omite alguno de los parámetros opcionales, se usan sus valores actuales. Así, si el color y la posición (X,Y) se omiten, el color utilizado para visualizar el *sprite* sería el color actual activo, y la posición donde se colocaría el *sprite* sería

la última accedida por un comando gráfico. La posición del *sprite* siempre está referida a la esquina superior izquierda del mismo. La posición (X,Y) puede también definirse en términos del comando STEP (X,Y), que ya empleamos en los comandos gráficos. Por tanto, para empezar situaremos un *sprite* en el centro de la pantalla. Antes de ver un programa que realice esta operación, apuntemos algo acerca del rango de valores permitido para las coordenadas X e Y.

La coordenada X debe estar comprendida entre −32 y 255; y la Y entre −32 y 191. Esto es válido en todos los modos de pantalla que soportan *sprites*, dando así un alto grado de control del posicionamiento de *sprites*. Algunos valores de Y dan resultados sorprendentes; si fijamos Y a un valor de 208, todos los *sprites* de los planos de *display* de prioridad más baja desaparecerán de la pantalla. Esta situación se mantiene hasta que es cambiado el valor de Y. Una valor de 209 origina la desaparición del mismo *sprite*.

De cualquier modo, colocaremos ahora nuestro *sprite* en la pantalla. Pruebe el siguiente programa:

```
10 SCREEN 2,1
20 FOR I=1 TO 8:READ n:a$+CHR$(n)
30 NEXT
40 SPRITE$(1)=a$
50 PUT SPRITE 1,(100,100),1
60 GOTO 60
70 DATA 56,56,16,56,84,16,40,68
```

La línea 10 establece el modo de pantalla y el tipo de *sprite* (en este caso se ha seleccionado el de 8 por 8 *pixels* ampliado). Este es posicionado en las coordenadas 100,100 de la pantalla, según la línea 50, y es pintado de negro. El número de plano en esta ocasión es 1, lo cual indica que el *sprite* definido en SPRITE$(1) aparecerá en esa posición. En circunstancias normales existe una relación directa entre el valor de *n* en el comando SPRITE$(*n*) y el plano de *sprite* en el que se visualizará en particular dicho *sprite*. Por ello la definición de *sprite* contenida en SPRITE$(1) aparecerá generalmente en el plano de *display* número 1. Para comprobarlo, añada al programa las siguientes líneas. En esta demostración el *sprite* definido en SPRITE$(2) tendrá prioridad más baja que el de SPRITE$(1).

```
35 FOR I=1 TO 8:READ n:b$=b$+CHR$(n):NEXT
45 SPRITE$(2)=b$
55 PUT SPRITE 2,(100,100),15
80 DATA 0,0,0,255,255,0,0,0
```

Por otra parte, ¿qué ocurre si queremos colocar el *sprite* definido en SPRITE$(2) en un plano de prioridad más alta que el del *sprite* definido en SPRITE$(1)? Aquí es donde se manifiesta la utilidad del parámetro de número de patrón en la instruc-

ción PUT SPRITE. Ejecute el programa listado a continuación y, al pulsar cualquier tecla, comprobará cómo podemos usar el número de patrón para posicionar un *sprite* en un plano distinto del que estaba asignado originalmente utilizando el comando SPRITE$:

```
10 SCREEN 1,1
20 FOR I=1 TO 8:READ n:a$=a$+CHR$(n):NEXT
30 FOR I=1 TO 8:READ n:b$=b$+CHR$(n):NEXT
40 SPRITE$(1)=a$:SPRITE$(2)=b$
50 PUT SPRITE 1,(100,100),1,1
60 PUT SPRITE 2,(100,100),15,2
70 a$=INPUT$(1)
80 PUT SPRITE 1,(150,150),1,2
90 PUT SPRITE 2,(150,150),15,1
100 a$=INPUT$(1)
110 GOTO 50
120 DATA 56,56,16,56,84,16,40,68
130 DATA 0,0,0,255,255,0,0,0
```

Una de las propiedades más provechosas de un *sprite* es que se le puede mover por la pantalla con el mínimo esfuerzo. La manera de hacerlo es simplemente cambiar las coordenadas X e Y. El *sprite* en cuestión será borrado entonces de su posición y colocado en la nueva según las coordenadas especificadas casi instantáneamente. En consecuencia, si sólo alteramos las coordenadas en, digamos, una unidad, podemos conseguir la sensación de movimiento continuo. El programa que exponemos a continuación mueve el *sprite* de la esquina superior izquierda de la pantalla a la inferior derecha.

```
10 SCREEN 1,1
20 FOR I=1 TO 8:READ n:a$=a$+CHR$(n):NEXT
30 SPRITE$(1)=a$
40 FOR I=0 TO 190
50 PUT SPRITE 1,(I,I),1
60 NEXT
70 GOTO 40
80 DATA 56,56,16,56,84,16,40,68
```

Si lo deseamos, podemos utilizar el comando ON INTERVAL GOSUB para actualizar automáticamente la posición del *sprite*. El programa siguiente demuestra esta operación:

```
10 ON INTERVAL=10 GOSUB 1000
30 SCREEN 1,1
40 FOR I=1 TO 8:READ n:a$=a$+CHR$(n):NEXT
```

```
50 SPRITE$(1)=a$
60 X%=128:Y%=96:XI%=1:YI%=1
65 INTERVAL ON
70 FOR I=1 TO 10000:PRINT I:NEXT
80 END
90 DATA 56,56,16,56,84,16,40,68
1000 PUT SPRITE 1,(X%,Y%),1
1010 X%=X%+XI%:Y%=Y%+YI%
1020 IF RND(2)*6>4 THEN XI%=-XI%
1030 IF RND(2)*4>3 THEN YI%=-YI%
1040 RETURN
```

Este programa es muy útil para demostrar el concepto de tiempo compartido, donde el ordenador está realizando aparentemente dos tareas a la vez. Es válido en aplicaciones donde se requiera hacer un trabajo extra, como dibujar un fondo gráfico, mientras los *sprites* continúan moviéndose por la pantalla.

No sólo podemos mover por la pantalla un *sprite,* sino varios. Para ello utilizaremos exactamente la misma clase de rutinas que hemos visto aquí, teniendo un par de coordenadas X,Y distintas para cada *sprite.* Por otra parte, si los mueve mediante el método ON INTERVAL, recuerde prolongar hasta consumir el tiempo de demora. Si no lo hace bien, puede encontrar al programa ejecutando continuamente la rutina de intervalo. También, cuando maneje a la vez varios *sprites* en la pantalla, recuerde que no puede tener al mismo tiempo más de cuatro en la misma línea de la pantalla en la dirección horizontal.

Coincidencia de "sprites"

Cuando un *sprite* se cruza en la pantalla con otro, se dice que son COINCIDENTES. Cuando diseñamos y programamos juegos gráficos es conveniente saber cuándo dos *sprites* coinciden en la pantalla; cuando ocurra, podemos tratarlo como una interrupción usando la sentencia ON SPRITE. Si un *pixel* de un *sprite* coincide en un cierto plano con un *pixel* de un *sprite* de un plano diferente, la CPU es informada en ese mismo momento por el VDP y, si está activa la instrucción ON SPRITE, se ejecuta inmediatamente la subrutina de tratamiento de interrupciones. El programa siguiente muestra esto. Observe el comando SPRITE OFF dentro de la subrutina: desactivará la interrupción por ON SPRITE hasta que se ejecute el siguiente comando SPRITE ON. Si este comando no está presente, tan pronto como coincidan los dos *sprites* se entrará en la rutina, se ejecutará y se regresará de ella. Sin embargo, si todavía coinciden los dos *sprites,* se entraría otra vez en la rutina casi inmediatamente, sin haberse realizado ningún otro proceso. Por eso el comando SPRITE OFF permite variar la posición de los *sprites* después de haberse detectado su colisión, y por tanto permite al programador continuar sin llamar repetidamente a la rutina. La rutina se activa de nuevo en la línea 160:

```
10 SCREEN 1,1:OPEN "GRP:" AS #1
20 FOR I=1 TO 8:READ n:a$=a$+CHR$(n):NEXT
30 SPRITE$(1)=a$:SPRITE$(2)=a$
40 X=100:Y=50:X1=100:Y1=70
50 X3=2:Y3=2:X2=2:Y2=2
60 ON SPRITE GOSUB 180
80 PUT SPRITE 1,(X,Y),1
90 PUT SPRITE 2,(X1,Y1),15
100 IF RND(2)*6>3 THEN X2=-X2
110 IF RND(2)*4>3 THEN Y2=-Y2
120 IF RND(2)*6>3 THEN X3=-X3
130 IF RND(2)*4>3 THEN Y3=-Y3
140 X=X2+X:Y=Y2+Y
150 X1=X1+X3:Y1=Y1+Y3
160 SPRITE ON
170 GOTO 80
180 BEEP:BEEP
190 PRINT #1,"- Ouch !"
200 SPRITE OFF
210 RETURN
220 DATA 56,56,16,56,84,16,40,68
```

Con lo expuesto terminamos el tema de cómo el programador de MSX puede usar las sentencias y comandos BASIC para controlar el procesador de *display* de video. Ya estamos preparados para ver cómo podemos ampliar las habilidades de nuestro ordenador MSX accediendo directamente al VDP y a la VRAM:

Acceso directo a la VRAM y al VDP

El BASIC MSX viene equipado con algunos comandos específicamente diseñados para acceder a la memoria de video y a los registros del VDP. Examinaremos primero estos comandos. Como ya hemos visto, la orden POKE se usa para modificar directamente los contenidos de memoria; de un modo similar utilizamos el comando VPOKE para alterar directamente los contenidos de la RAM de video. La sintaxis completa es

```
VPOKE direccion,valor
```

donde la dirección está entre 0 y 16384, y el valor, entre 0 y 255. Usamos la instrucción VPEEK (dirección) para conocer el contenido de una dirección particular en la memoria de video, estando de nuevo la dirección entre 0 y 16384.

Para tener acceso directo a los registros del procesador de *display* de video (VDP) utilizamos el comando VDP(*n*), donde *n* es un número de registro entre 0 y 8. La sentencia VDP nos permite ver el valor actual de los registros sólo de escritura y también alterar sus contenidos. También podemos usarlo para conocer el valor de los registros sólo de lectura del VDP. El comando

```
PRINT VDP(1)
```

imprimirá en la pantalla el valor actual contenido en el registro 1 del VDP. Si deseamos asignar un valor al registro usaremos la instrucción siguiente.

```
VDP(1)=valor
```

Esto fijará el registro al valor situado a la derecha del signo =. Una advertencia: ciertos registros pueden provocar que el VDP se comporte de una manera muy peculiar si toman ciertos valores. No causará ninguna avería en el ordenador, pero ¡puede que tenga que hacer un *reset* para recuperar el control de la máquina! De cualquier modo, veamos la función de cada registro del VDP y aprendamos cómo podemos usarlo para ampliar nuestras técnicas de programación.

Notas generales sobre los registros del VDP

Antes de proseguir con la programación de registros es aconsejable que consulte el apéndice sobre sistemas de numeración, prestando una atención especial al sistema binario. Si lo hace, entenderá mejor esta sección del libro, Los registros del VDP pueden contener cada uno un número de 8 bits, que pueden tomar un valor entre 0 y 255; este número contenido en el registro puede tratarse como indicador de una dirección en VRAM o como una serie de bits que controlan cada uno alguna faceta de la operación del VDP. Si el contenido de un registro se usa para controlar la operación del VDP, y solamente queremos alterar un bit del registro, debemos tener cuidado de no alterar el resto accidentalmente.

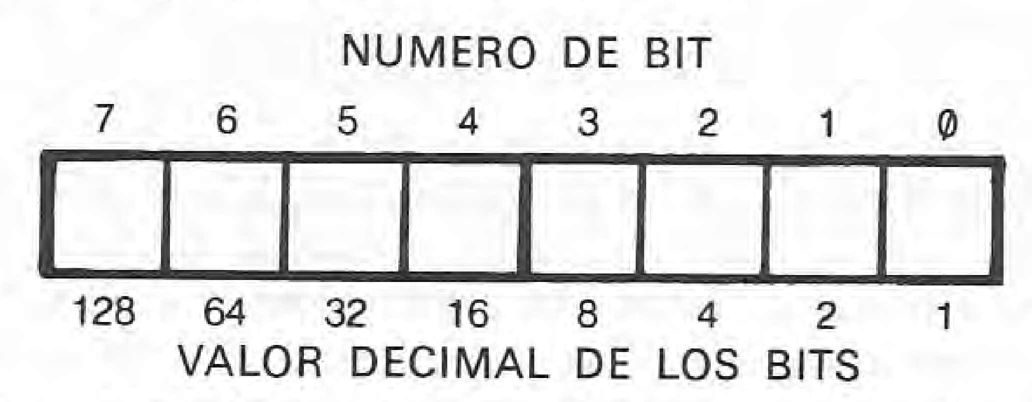


Figura 6.10.—*Los registros del procesador de* display *de video.*

Cada posición del bit tiene, como ya sabrá si ha leído el apéndice, un valor numérico asociado. Estos valores son los expuestos en la figura 6.10. Para evitar alterar los otros bits del registro cuando cambiamos uno, empleamos las operaciones binarias AND y OR, que vimos en el capítulo 3. Por ejemplo, para fijar el bit 3 del registro *n* a 1, dejando invariables los otros bits, utilizamos la línea:

```
VDP(n)=VDP(n) OR &B00001000
```

Se ha usado el sistema binario para hacer la operación OR más comprensible.

De manera similar, para fijar el mismo bit a 0 utilizamos la operación binaria AND:

```
VDP(n)=VDP(n) AND &B11110111
```

Esto es bastante importante cuando trabajamos, por ejemplo, con el registro 1 del VDP, que está totalmente dedicado a controlar el *chip*. Si desea comprobar cuándo un bit concreto de un registro es 1 ó 0, de nuevo podemos utilizar los operadores binarios, como vimos en el capítulo 3.

Registro 0

Sólo se utilizan, por el momento, los bits 0 y 1 de este registro; el resto de los bits se reservan por el fabricante para un uso futuro. De estos dos bits, el único relevante para el programador MSX es el 1, que está implicado en la selección del modo de pantalla. Este bit se llama M3, y veremos cómo se emplea cuando estudiemos el registro 1.

Registro 1

Es el registro de control principal del VDP. Algunos bits son bastante útiles, mientras que un par de ellos es mejor dejarlos aparte.

Bit 7. Es uno de los últimos; dice al VDP qué *chips* se usan para la memoria de video. Si altera el valor que su ordenador particular necesita, la pantalla se volverá ilegible.

Bit 6. Controla si la pantalla está en blanco o si está activa. Si lo colocamos a 0 se apagará la pantalla y la única parte visible será el color de borde, que ahora será visto por todo el *display*. Este bit está fijado habitualmente a 1.

Bit 5. Este bit se llama bit de ACTIVACION DE INTERRUPCION DEL VDP, y requiere un cuidado especial. En los ordenadores MSX, el VDP genera una señal de interrupción, cada 1/50 de segundo; en las máquinas europeas esto provoca que la CPU realice varios trabajos "rutinarios", como leer el teclado, actualizar la variable TIME y los contadores usados en el comando ON INTERVAL. Si este bit se fija a 0, se dice que la interrupción está desactivada, y ninguna de estas tareas será llevada a cabo por la CPU. Por tanto, fijando este bit a 0 se evita la operación de lectura del teclado. ¡No debe hacerse desde el modo directo! La única manera de recuperar el control del ordenador es haciendo un *reset*. Sin embargo, puede hacerse desde dentro de un programa, y éste proseguiría su ejecución sin considerar el estado de la interrupción. Obviamente el programa no debe necesitar ninguna entrada desde el teclado, ya que provocaría la espera de una entrada de datos que nunca llegaría. Dos posibles beneficios pueden obtenerse de desactivar esta interrupción.

- i Si desactiva esta interrupción antes de entrar en un bucle muy largo, que no contenga entradas de datos o referencias a la variable TIME, causará que se acorte ligeramente el tiempo de ejecución del bucle. Recuerde volver a colocar en 1 este bit antes de seguir con el programa.

ii Si desea una protección de programa astuta, puede emitir un comando de desactivación de la interrupción como parte de una rutina de tratamiento de interrupciones (ON STOP). ¡Esta es una manera extrema de prevenir que el usuario vea su programa!

La interrupción es activada por

```
VDP(1)=VDP(1) OR &B00100000
```

y desactivada por el comando

```
VDP(1)=VDP(1) AND &B11011111
```

Bits 4 y 3. Junto con el bit M3 del registro 0, estos dos bits controlan el modo de pantalla seleccionado por el VDP. El bit 4 se llama M2, y el 3, M1. El modo de pantalla seleccionado depende de los valores de estos tres bits.

M1	M2	M3	Modo de pantalla
0	0	0	1
0	0	1	2
0	1	0	3
1	0	0	0

Bit 1. Selecciona el tipo de *sprite* a usar. Si es 0 será seleccionado el *sprite* de 8 por 8 *pixels*. Si es 1 será el más grande, el de 16 por 16 *pixels*.

Bit 0. Selecciona si los *sprites* están ampliados o no; si es 0, los *sprites* no serán ampliados, y si es 1 sí lo serán.

El resto de los registros del VDP, con excepción del de estado, contienen datos acerca de la posición de las distintas tablas de información contenidas en la VRAM y que el VDP necesita para realizar su funciones.

Registro 2

Contiene un valor entre 0 y 15. El contenido, multiplicado por &H400 nos dará el principio de lo que se llama la TABLA DE NOMBRES para ese modo de pantalla en concreto. Esto se estudiará con más detalle cuando veamos las direcciones de la VRAM dentro de poco tiempo.

Registro 3

Este registro contiene un valor entre 0 y 255 y, multiplicado por &H40, da la dirección en VRAM de lo que se llama TABLA DE COLORES para un modo particular.

Registro 4

Define la dirección de comienzo del área en VRAM de la TABLA DE PATRONES para un modo dado. Para conseguir esta dirección de VRAM el contenido del registro se multiplica por &H800.

Registro 5

El contenido de este registro, multiplicado por &H80, da la dirección en VRAM de la TABLA DE ATRIBUTOS DE *SPRITES*.

Registro 6

El contenido de este registro, multiplicado por &H800, da la dirección de comienzo en VRAM de la TABLA GENERADORA DE PATRONES DE *SPRITES*.

Registro 7

Este registro es el que más se usa en el modo de pantalla 0. Los 4 bits más altos del registro contienen el código de color del texto activo en modo 0. Los 4 bits más bajos contienen el código del color de fondo en el modo 0, y el color del borde, en los otros modos. Los códigos utilizados para representar cada color son los mismos que usamos en el comando COLOR. Pruebe el programa siguiente, que visualizará todas las combinaciones posibles de color de fondo y del texto. Recuerde que con algunas de estas combinaciones la pantalla será ilegible.

```
10 SCREEN 0
20 PRINT "Hola"
30 FOR I=0 TO 255
40 VDP(7)=I
50 TIME=0
60 IF TIME<20 THEN GOTO 60
70 NEXT
80 SCREEN 0
```

La línea 80 fijará los códigos de color contenidos en el registro a sus valores normales; en este caso, el valor tomado por defecto es de 244.

El registro de estado

Es el registro sólo de lectura del VDP, y no podemos alterar su valor directamente. De la única forma que este registro cambia su valor es respondiendo a cam-

bios en la operación del VDP. Por eso sólo puede accederse a él por medio de sentencias como

```
var=VDP(8)
```

o

```
PRINT VDP(8)
```

hay 3 bits del registro que se usan para señalar a la CPU cambios en el estado del VDP. Estos bits se llaman BANDERAS *(flags)*.

Bit 7. Sólo tiene utilidad real para el programador de código máquina. Indica que el VDP quiere enviar una interrupción a la CPU. Esta interrupción sólo será enviada si el bit de activación de interrupción está a 1; si está a 0 no se envía. Si escribe programas que utilicen este bit, solamente se pondrá a 0 leyendo el registro de estado. Si no lo lee, el VDP no será capaz de indicar cuándo quiere mandar su próxima interrupción.

Bit 5. Se llama bandera de COINCIDENCIA; está a 1 siempre que coinciden dos *sprites,* y sólo se pone a 0 por una lectura del registro de estado. Es, también, de uso real sólo para el programador en código máquina, ya que el BASIC MSX hace uso total de esta facilidad con el comando ON SPRITE. En las máquinas MSX europeas, que usan el sistema de TV PAL, el VDP comprueba la coincidencia de *sprites* cincuenta veces por segundo. Como solamente se pone a 0 leyendo el registro de estado, cualquier programa en código máquina que use esta facilidad debe leer el registro cada cierto tiempo para asegurarse de que la bandera no está bloqueada.

Bit 6. Se llama la bandera del QUINTO SPRITE; no hace mucho se dijo que sólo podíamos tener 4 *sprites* por línea horizontal de la pantalla; esta bandera se fija en 1 siempre que una línea está a punto de contener un quinto *sprite.* El número del *sprite* que está causando el problema (el número de plano de *sprite,* para ser precisos) se almacena en ese momento en los bits 0 a 4 del registro.

La manera de acceder a los registros del VDP a través del código máquina será discutida en el capítulo 11.

Uso de la VRAM en los modos de pantalla

El sistema MSX tiene 16K de memoria dedicada totalmente al VDP. Se llama VRAM, y la manera en que se distribuye por ella la información depende del modo de pantalla en funcionamiento. La memoria usada por un modo dado se utiliza para varios propósitos, y cada área de RAM tiene un nombre. La TABLA DE NOMBRES es un área de VRAM que indica al VDP qué imagen debe aparecer en una cierta parte de la pantalla. La manera en que los contenidos de la tabla de nombres se convierten en imágenes visuales depende del modo, como veremos pronto. La TABLA DE GENERACION DE PATRONES es el área de RAM que define la imagen a colocar en la pantalla para un valor dado de la tabla de nombres. La TABLA DE COLORES informa al VDP de los colores a usar en ese modo. La TABLA DE ATRIBUTOS

DE *SPRITES* y la TABLA DE GENERACION DE *SPRITES* se estudiarán más adelante, en este mismo capítulo. Cuando conectamos nuestro ordenador, la ROM de la máquina fija las direcciones de comienzo para cada tabla según el modo a usar. Las direcciones de comienzo apropiadas se colocan entonces en los registros correctos del VDP. Podamos localizar estas direcciones en la VRAM usando una instrucción llamada BASE. Esta sentencia se usa para leer la dirección de comienzo de las distintas tablas en un modo de pantalla dado, y su sintaxis completa se expone a continuación:

```
comienzo=BASE(n)
```

donde *n* es un valor entre 0 y 19. Algunos valores de *n* dentro de este rango retornarán resultados sin sentido, pero los valores válidos de *n* se mencionarán cuando examinemos los distintos modos. Estas direcciones de comienzo de tablas pueden alterarse escribiendo en el correspondiente registro del VDP, y no mediante el comando BASE. Ahora empecemos con los modos de pantalla.

Modo 0

Este es el modo de texto de 24 líneas y 40 caracteres por línea. Solamente se usan dos tablas, ya que los colores usados en este modo se almacenan en el registro 7 del VDP. BASE(0) nos facilitará la dirección de comienzo de la tabla de nombres para este modo, y BASE(2) nos indicará la de la tabla de patrones.

La tabla de nombres contiene información que utiliza el VDP para presentar en la pantalla una imagen definida en cierta posición de la tabla de patrones. El modo de operar de esta tabla es el mismo en cada modo de pantalla, aunque su longitud puede ser diferente. Los bytes de la tabla de nombres corresponden a posiciones de pantalla, y los contenidos de la tabla se usan para acceder a la tabla de patrones en cada modo.

En modo 0 la tabla de nombres tiene 960 bytes de largo, y está relacionada con la pantalla según se muestra en la figura 6.11. Así, el valor contenido en la posición 20 de la tabla de nombres definirá la imagen visualizada en la fila 1, columna 20 de la pantalla.

El programa que se expone a continuación escribirá cada posición de la tabla de nombres, usando el comando VPOKE. A cada posición se le asigna el número 35, que es el código ASCII de "#". Los números de la tabla de nombres corresponden a los códigos ASCII de los caracteres, y por eso este programa llena la pantalla de signos "#".

```
10 SCREEN 0
20 comienzo=BASE(0)
30 FOR I%=comienzo TO comienzo+960
40 VPOKE I%,35
50 NEXT
```

```
60 I$=INPUT$(1)
70 CLS
```

TABLA DE NOMBRES (MODO 0)

Base (Ø)	BYTE Ø
	BYTE 1
	BYTE 2
	BYTE 3
	BYTE 4
	BYTE 5
	⋮
	BYTE 958
Base (Ø)+959	BYTE 959

PANTALLA EN MODO 0

Ø	1	2	3	…	38	39
4Ø	41	42	43	…	78	79
8Ø	81	82	83	…	118	119
⋮	⋮	⋮	⋮		⋮	⋮
92Ø	921	922	923	…	958	959

Figura 6.11.—*Relación entre la pantalla y la tabla de nombres, modo 0.*

Cuando la pantalla esté llena, pulse cualquier tecla para seguir. También es posible saber qué carácter es visualizado en cualquier punto de la pantalla usando VPEEK, para consultar la dirección apropiada de la tabla de nombres. En modo 0, los bytes de la tabla de nombres se refieren a posiciones de texto, porque de hecho contienen códigos de caracteres.

La tabla de patrones contiene los datos necesarios para que el VDP convierta el código ASCII de carácter contenido en la tabla de nombres en una imagen en la pantalla, como el signo "#". Para cada carácter del sistema ASCII se necesitan 8 bytes para almacenar la imagen de pantalla. Hay 256 códigos ASCII distintos, y la tabla de patrones tiene longitud suficiente para incluir todos estos códigos. Consta de 2.048 bytes en modo 0. Los primeros 8 bytes de la tabla contienen información para la imagen en pantalla del carácter con el código ASCII 0; los 8 siguientes, los datos para el carácter 1, y así, hasta los últimos 8 bytes a partir de la posición 2040 de la tabla, que contiene los datos del carácter 255. Los datos para la imagen de pantalla se almacenan de modo idéntico a como definimos los *sprites;* el primero de los 8 bytes que representa un carácter dado nos proporciona los datos para la fila 1 del carácter; el segundo byte codifica la segunda fila, etc.

La mejor manera de explorar la tabla de patrones es intentar alterar los caracteres normalmente visualizados. Alteremos carácter "espacio" de alguna manera. Bien, lo primero que tenemos que hacer es averiguar dónde empieza la definición del carácter "espacio" en la tabla de patrones. Esto es bastante fácil;

```
espacio=BASE(2)+(32*8)
```

devolverá el comienzo de la definición del carácter "espacio": Como método más general, la línea siguiente nos dará la dirección de comienzo en VRAM de la definición del carácter con código ASCII *n*.

```
comienzo=BASE(2)+(n*8)
```

Esto es, desde luego, aplicable solamente en modo 0. Esta dirección será la de la primera fila de la definición del carácter.

El programa siguiente modificará la primera línea de la definición del carácter "espacio":

```
10 SCREEN 0
20 comienzo=BASE(2)
30 VPOKE (comienzo+(32*8)),255
```

Ejecute este programa. Observará que la pantalla aparece rayada. Si en el comando VPOKE pusiéramos la dirección de comienzo +(32*8)+7 la línea aparecería en la parte de abajo de cada espacio de la pantalla. Cualquier carácter puede redefinirse de esta manera en el modo 0. Como ejemplo, redefinamos el carácter con el código ASCII 254, con el bloque de la figura 6.12.

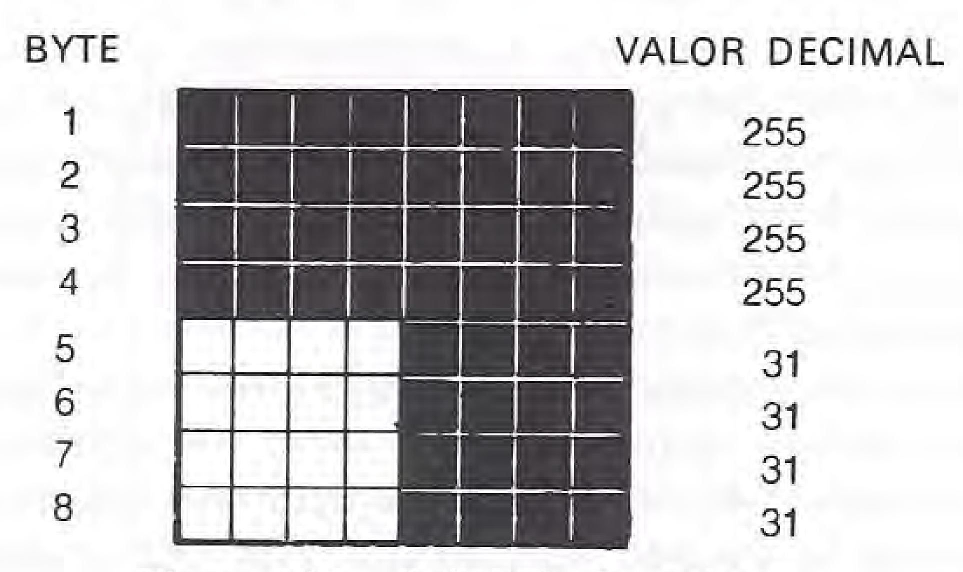


Figura 6.12.—*Definición de carácter.*

Siga el procedimiento de la matriz detallado en la sección de este capítulo concerniente a la definición de *sprites*. Haciéndolo así, la sentencia DATA del programa siguiente presenta los resultados. Observe que en este modo los caracteres parece que están basados en una matriz de 8 filas por 6 columnas; las 2 columnas situadas más a la derecha de la matriz no se consideran a efectos de definición.

```
10 SCREEN 0
20 FOR I=0 TO 7:READ n
30 VPOKE (BASE(2)+((254*8)+I)),n
40 NEXT
50 DATA 255,255,255,255,31,31,31,31
```

Ejecute el programa y teclee PRINT CHR$(254). Si todo ha ido bien, su pequeño bloque será visualizado en la pantalla. Este nuevo CARACTER DEFINIDO POR EL USUARIO puede ahora ser tratado como si fuera un carácter estándar ASCII. Sólo hay un inconveniente cuando se definen caracteres de este modo: si ejecutamos la orden SCREEN 0, el ordenador limpia automáticamente la tabla de nombres, borrando así la pantalla. También reinicializa todas las definiciones de caracteres de la tabla de patrones, utilizando para ello los caracteres contenidos en la ROM del sistema. Por ello, después de cada cambio de modo, cualquier carácter especial ha de ser redefinido. Los colores en que aparecerá su nuevo carácter serán los mismos que los del resto de los caracteres. Cualquier "1" en sus bytes de definición será del color del texto, y los "0" serán del color de fondo.

Alterando los contenidos del registro apropiado del VDP, es posible tener a la vez en memoria dos tablas de patrones alternativas. Por eso puede tener juego alternativo de caracteres e intercambiarlos alterando el valor del registro 4 del VDP. Se puede hacer algo similar con la tabla de nombres, permitiéndonos elegir entre dos pantallas de datos alternativamente. El programa siguiente expone esta operación. Podría ser ampliamente desarrollado, pero el propósito de la rutina es mostrar los principios implıcados. Fijamos el registro 2 del VDP a 12. Entonces llenamos esta área de RAM con el comando VPOKE. Simplemente cambiando el valor contenido en el registro 2 se visualizan alternativamente en la pantalla las dos diferentes tablas de nombres:

```
10 SCREEN 0
20 VDP(2)=12:comienzo=12*1024
30 FOR I%=comienzo TO comienzo+960:VPOKE I%,65
40 NEXT
50 PRINT "Hola"
60 VDP(2)=0:REM valor tomado por defecto
70 I$=INPUT$(1)
80 VDP(2)=12
90 I$=INPUT$(1)
100 GOTO 60
```

La línea 50 nos demuestra que podemos escribir información en la pantalla con el comando PRINT mientras visualizamos el contenido de la tabla de nombres alternativa; esto es así porque las rutinas en que manejan la escritura en la pantalla lo hacen en posiciones específicas dentro de la VRAM. Pulsando una tecla elegiremos entre las dos tablas de nombres y, por tanto, cambiaremos los contenidos de la pantalla.

Modo 1

Otro modo de texto, pero con tres tablas. Estas son las de nombres y de patrones, que realizan tareas similares a las del modo 0, y la tabla de colores, que contiene información acerca de los colores a visualizar. El comienzo de la tabla de

nombres viene dado por BASE(5); el de la tabla de patrones, por BASE(7), y el de la de colores, por BASE(6). También hay dos tablas en VRAM relacionadas con los *sprites*, pero las consideraremos más tarde en este capítulo. Por el momento es suficiente decir que el comienzo de la TABLA DE ATRIBUTOS DE *SPRITES* lo da por BASE(8), y el de la TABLA DE PATRONES DE *SPRITES*, por BASE(9).

La tabla de nombres en este modo está relacionada con la pantalla de una manera análoga a como lo estaba en modo 0. Aquí, sin embargo, sólo tiene 768 bytes de largo, y puede accederse a ella de manera similar al modo 0, es decir, utilizando VPOKE y VPEEK. De hecho, en el programa de demostración: alterando directamente los valores en memoria de la tabla de nombres en modo 0 funcionará en modo 1; alterando BASE(0) y cambiando por BASE(5); reemplazando 960 por 768 y sustituyendo SCREEN 0 por SCREEN 1. Podemos alterar su dirección de comienzo alterando el registro 2 del VDP en este modo.

La tabla de patrones realiza también la misma función que en el modo 0, y podemos redefinir caracteres en este modo 1 de la misma manera que lo hacíamos en aquél. Simplemente usamos BASE(7) en lugar de BASE(2) para obtener la dirección de comienzo de la tabla de patrones.

La principal diferencia en este modo es la presencia de la tabla de colores, que incrementa su disponibilidad para el programador MSX. En los manuales se dice que en modo 1 sólo hay dos colores disponibles; pero esto no es totalmente cierto, como ahora veremos. La tabla de colores solamente tiene 32 bytes de largo. Su dirección de comienzo en VRAM puede alterarse dando un valor diferente al registro 3 del VDP, mientras se esté en este modo. Cada byte de la tabla de colores contiene un valor relacionado con los usados en ocho caracteres definidos en la tabla de patrones. Los 4 bits más altos de la tabla de colores contienen el color del carácter, y los 4 más bajos, el color a usar como fondo para estos ocho caracteres. El primer byte de la tabla de colores contiene la información de color para los primeros 8 bytes definidos en la tabla de patrones; o sea, los códigos ASCII de 0 a 7. El segundo byte define los colores para los códigos ASCII de 8 a 15, etc. Es decir, el byte número 32 de la tabla contiene la información de color para los caracteres del 248 al 255. Por ello, es posible cambiar los valores de cada posición de la tabla de tal manera que podamos tener los ¡dieciséis colores disponibles a la vez! en las máquinas MSX en el modo de pantalla 1. El inconveniente de este método es que los colores están "ligados" a ciertos caracteres, pero con una selección cuidadosa de las posiciones de la tabla de colores pueden obtenerse algunos efectos bastante sorprendentes.

El programa siguiente enseña a alterar los colores empleados en los caracteres con código ASCII entre 64 y 71:

```
10 SCREEN 1
20 comienzo=BASE(6)
30 VPOKE (comienzo+8),18
40 END
```

Ejecute el programa y lístelo en la pantalla. Bonito, ¿verdad?

En modo 1 se utilizan 2.848 bytes en VRAM para las tablas de nombres, patrones y colores. Si se quiere usar *sprites*, se necesita VRAM adicional.

Modo 2

La manera en la que se organiza la VRAM en este modo es algo diferente a como lo hace en los dos modos de texto que ya hemos visto. Esto era de esperar, debido a la capacidad de alta resolución del modo 2. Es algo bastante complejo de entender, pero avanzaremos lentamente para que usted sea capaz de comprender la asignación de VRAM en modo 2. La dirección de comienzo de la tabla de nombres viene dada por BASE(10); la dirección de comienzo de la tabla de colores, por BASE(11), y la de la tabla de patrones, por BASE(12). La dirección de comienzo de la tabla de atributos de *sprites* viene dada por BASE(13), y la de la tabla de patrones de *sprites*, por BASE(14).

La tabla de nombres aquí también tiene solamente 768 bytes de largo, y la colocación del mapa de esta tabla sobre la pantalla se produce de la misma manera que hemos visto en los modos 0 y 1. No obstante, la diferencia es que la imagen mostrada en una posición de la pantalla no sólo depende del contenido de la posición correspondiente en la tabla de nombres, sino también de en qué lugar de ésta —y por tanto de la pantalla— ha de colocarse la imagen. Esto parece un poco complicado, pero no se preocupe; intentaremos explicarlo con más claridad en su momento.

La tabla de patrones en este modo ocupa 6.144 bytes; o sea, es tres veces mayor que la tabla de patrones de cualquiera de los otros dos modos que hemos visto, y permite que cada posición de la tabla de nombres tenga un código totalmente único, posibilitando así que cada posición de pantalla sea diferente de cualquier otra. La tabla de colores también tiene disponibles 6.144 bytes, y permite a cada posición de la tabla de nombres tener para ella una combinación de colores totalmente única. Es el tamaño de estas dos tablas el que nos da la capacidad para gráficos en modo 2.

La pantalla y las tablas de memoria relacionadas con ella es mejor tratarlas dividiéndolas en tres partes. El diagrama de la figura 6.13 lo muestra.

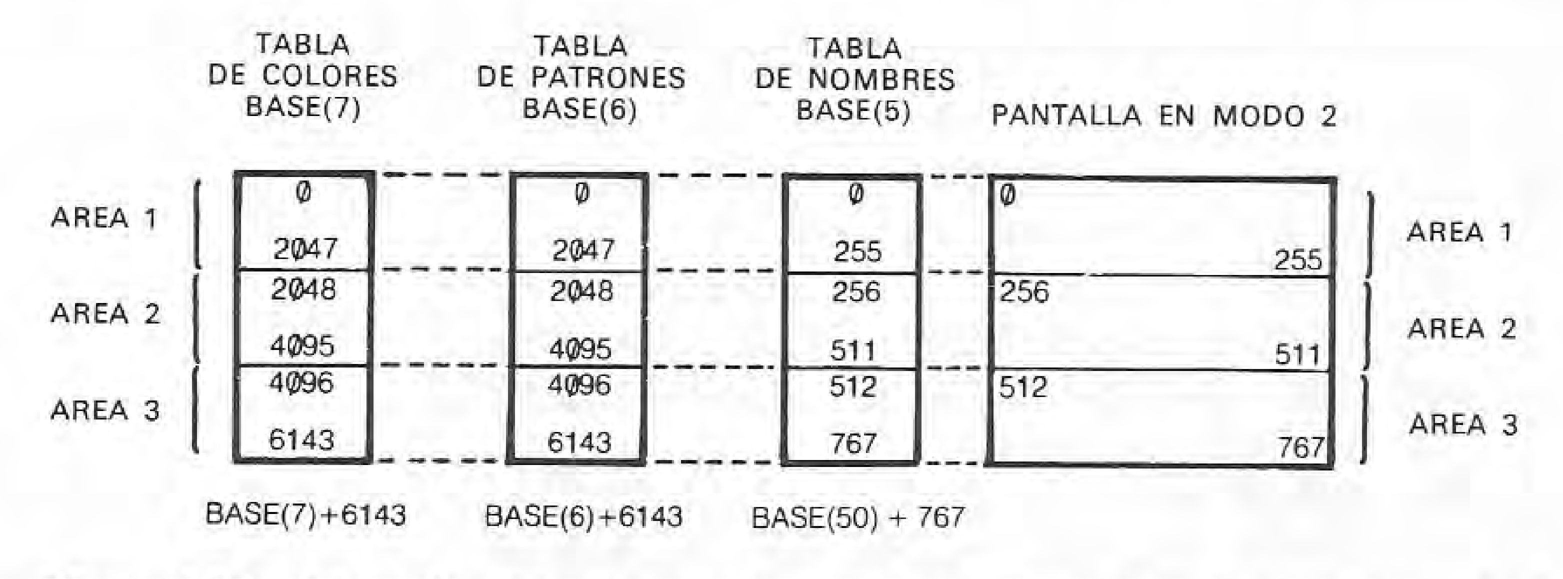


Figura 6.13.—*Mapa de tablas de colores, patrones y nombres en la pantalla, modo 2.*

La tabla de patrones, a pesar de ser mucho más grande, está organizada de manera similar a la tabla de patrones del modo 0: se divide en bloques de 8 bytes definiendo cada bloque un patrón que puede aparecer en la pantalla como una imagen. Los primeros 2.048 bytes de la tabla de patrones pueden por ello verse como 256 bloques de 8 bytes, definiendo cada uno de ellos la imagen de las 256 posiciones de la tabla de nombres en el área 1, y por tanto definiendo la imagen que aparecerá en la correspondiente posición de la pantalla. Similarmente, los siguientes 2.048 bytes hacen el mismo trabajo para el área 2 de la tabla de nombres, y los 2.048 finales sirven para el área 3 de la tabla de nombres. Así, si el valor de la posición 200 de la tabla de nombres, que corresponde a la posición de pantalla 200, era el número 2, la imagen que sería visualizada en esa posición de pantalla estaría definida por los bytes 16 a 23 del área 1 de la tabla de patrones. De una manera similar, si la posición 512 de la tabla de nombres, que asimismo corresponde a la posición de pantalla 512, tiene el valor 0, la imagen visualizada en esa posición de pantalla estaría definida por los primeros 8 bytes del área 2 de la tabla de patrones; o sea, los bytes 2.048 a 2.057 de la tabla de patrones tratada como un todo. Existen relaciones similares entre la tabla de nombres y la tabla de colores.

Cada byte de la definición de patrón puede tener dos colores; los 4 bits más altos de la posición correspondiente en la tabla de colores dan el color del primer término de este byte particular del patrón, y los 4 bits más bajos de la posición correspondiente de la tabla indican el color de fondo para ese byte de patrón definido.

Veamos ahora un par de ejemplos de acceso directo a la VRAM en modo 2. El primer ejemplo llena el tercio superior de la pantalla con un patrón. Digamos que el código que ha de aparecer en la tabla de nombres en el lugar donde queramos que aparezca el carácter es 0. Como estamos trabajando en el área 1 de la pantalla, debemos poner los datos que definen nuestro carácter en las primeras ocho posiciones de la tabla de patrones. De manera similar, los códigos de colores de este carácter en concreto deben ir en las primeras ocho posiciones de la tabla de colores.

En el programa que se expone a continuación, la línea 30 define el carácter según la sentencia DATA. La línea 50 fija los códigos de color para el carácter. Si le gusta experimentar, puede dar diferentes valores a cada byte a utilizar de la tabla de colores ¡para obtener un carácter en tecnicolor! Finalmente, la línea 70 del programa coloca el código 0 en las primeras 255 posiciones de la tabla de nombres, llenando por tanto el tercio superior de la pantalla con nuestro carácter.

```
10 SCREEN 2
20 FOR I=BASE(12) TO BASE(12)+7
30 READ N:VPOKE I,N:NEXT
40 FOR I=BASE(11) TO BASE(11)+7
50 VPOKE I,18:NEXT
60 FOR I=BASE(10) TO BASE(10)+255
70 VPOKE I,0:NEXT
80 GOTO 80
90 DATA 56,56,16,56,84,16,40,68
```

Usted debería ser capaz de ampliar este programa de tal manera que el hombrecillo aparezca por toda la pantalla usando la información dada anteriormente.

Un dato a destacar es que, una vez ha definido un carácter de la tabla de patrones, el carácter definido aparece inmediatamente en la pantalla en la posición apropiada. Por ello, colocar una definición en los bytes 0 a 7 de la tabla de patrones provocará que el carácter así definido aparezca en la posición de pantalla 0. Se observa un comportamiento similar con los colores. Para comprobarlo, ensaye el programa siguiente:

```
10 SCREEN 2
20 FOR I=BASE(12) TO BASE(12)+7
30 READ N
40 VPOKE I,N:VPOKE (I+2048),N
45 VPOKE (I+4096),N
50 NEXT
60 VPOKE BASE(10),0:GOTO 60
70 DATA 255,255,255,255,0,0,0,0
```

Este programa colocará la marca en la pantalla para indicar las posiciones 0, 256 y 512.

Se puede programar el movimiento accediendo directamente a la VRAM, como corrobora el siguiente programa. Las bases son bastante simples: fijamos en la tabla de nombres un valor que corresponde a la definición de imagen requerida en la tabla de patrones en esa área de la pantalla. Entonces hacemos una pausa y, seguidamente, fijamos en esa posición un código sin patrón asociado en la tabla de patrones. Tras una corta demora repetimos el proceso para la posición siguiente, y lo reiteramos análogamente:

```
10 SCREEN 2:REM reinicializa el VDP
20 FOR I=BASE(12) TO BASE(12)+7:READ n
30 VPOKE I,n:NEXT
40 FOR I=BASE(11) TO BASE(11)+7
50 VPOKE I,18:NEXT
60 FOR I=BASE(10) TO BASE(10)+255
70 VPOKE I,0:FOR J=1 TO 30:NEXT
80 VPOKE I,1:FOR J=1 TO 30:NEXT
90 NEXT
100 GOTO 60
110 DATA 56,56,16,56,84,16,40,68
```

El patrón 1 de la tabla de patrones se supone que es un carácter espacio; por ello borrará el hombrecillo, definido como patrón 0, cuando éste haya pasado a la posición siguiente.

La versatilidad del modo 2 en términos de sus gráficos de alta resolución se refleja en la cantidad de memoria ocupada en este modo por las tablas de patrones, nombres y de colores. Se ocupan 13.056 bytes de RAM de pantalla. Esto todavía, sin embargo, nos deja bastante espacio para diseñar una segunda tabla de nombres en VRAM, alterando el registro 2 del VDP para este modo. Al hacerlo, tenga cuidado de no escribir sobre parte de las tablas de colores o patrones.

Modo 3

Este es el modo de pantalla final del sistema MSX, y nos facilita gráficos de baja resolución con los dieciséis colores. BASE(15) da la dirección de comienzo de la tabla de nombres, y BASE(17), la de la tabla de patrón. BASE(18) y BASE(19) dan las direcciones de comienzo de las tablas de atributos y de *sprites*, respectivamente.

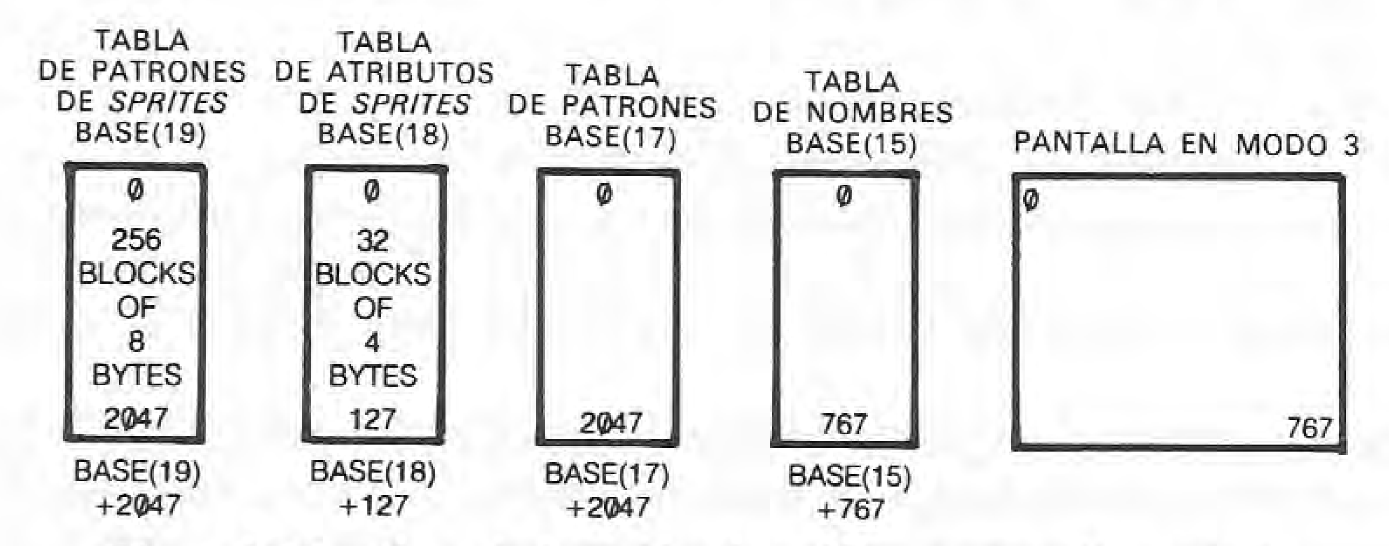


Figura 6.14.—*Mapa de las tablas de gráficos en pantalla, modo 3.*

Observando lo expuesto le llevará a la conclusión de la ausencia de una tabla de colores. Esto es porque en este modo la tabla de patrones contiene datos para las imágenes a visualizar, así como de los colores en que deben ser visualizadas.

La tabla de nombres en este modo es nuevamente de 768 bytes de largo; cada posición en esta tabla actúa como un puntero de un área de 8 bytes de la de patrones, pero no de la de colores. La tabla de patrones está organizada de una manera bastante peculiar para codificar a la vez la forma y el color de la imagen. Tiene 2.048 bytes de largo.

En modo 2 podemos trazar *pixels* aislados en la pantalla; la resolución del modo 3 no es tan buena; en realidad, solamente nos permite trazar "super-*pixels*", que son dieciséis veces un *pixel* normal. Estos pueden ser del color que desee, y, debido a la reducida resolución, cada matriz del carácter en la pantalla necesita solamente 2 bytes para estar totalmente definido, como se muestra en la figura 6.15. Sin embargo, ya hemos dicho que una posición de la tabla de nombres direcciona a un bloque de 8 bytes en la de patrones. Por tanto, ¿qué ocurre con los otros 6 bytes? Buscaremos la respuesta a esta pregunta enseguida.

Los 4 bits más altos del byte 1 contienen el código del color para el super-*pixel* 1; los 4 bits más bajos el código, para el super-*pixel* 2; los 4 más altos del segundo byte, el código de color del super-*pixel* 3, y los más bajos de ese mismo byte, el color del super-*pixel* 4.

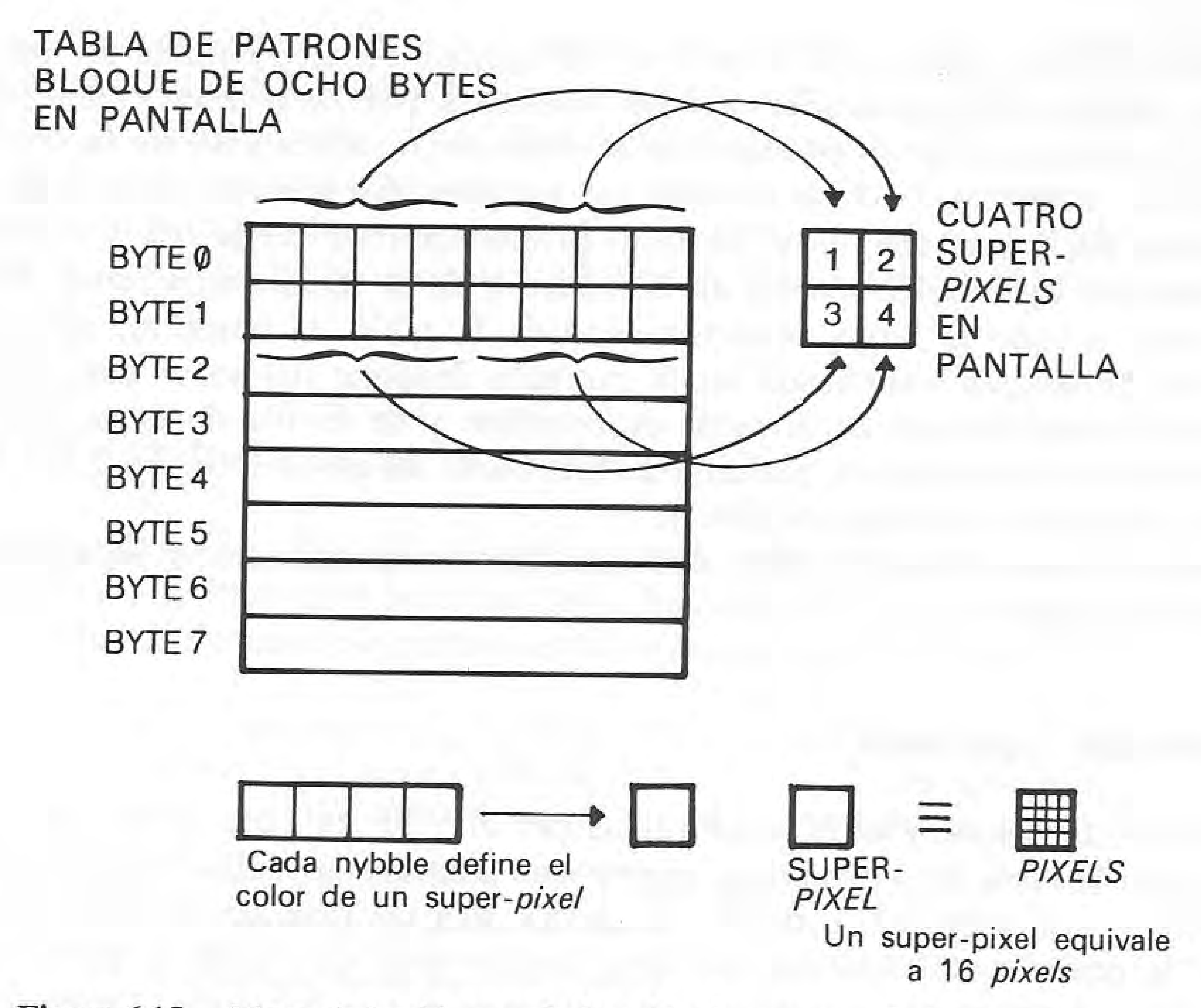


Figura 6.15.—*Mapa de los bloques de la tabla de patrones de la pantalla (1).*

Ahora veamos los 6 bytes que sobraban. Están divididos en tres grupos similares de 2 bytes, organizados como antes. ¿Qué 2 bytes de la definición de patrón se utilizan para visualizar en la pantalla? Depende de la fila donde esté colocada la posición concreta de la tabla de nombres. La figura 6.16 ilustra lo expuesto.

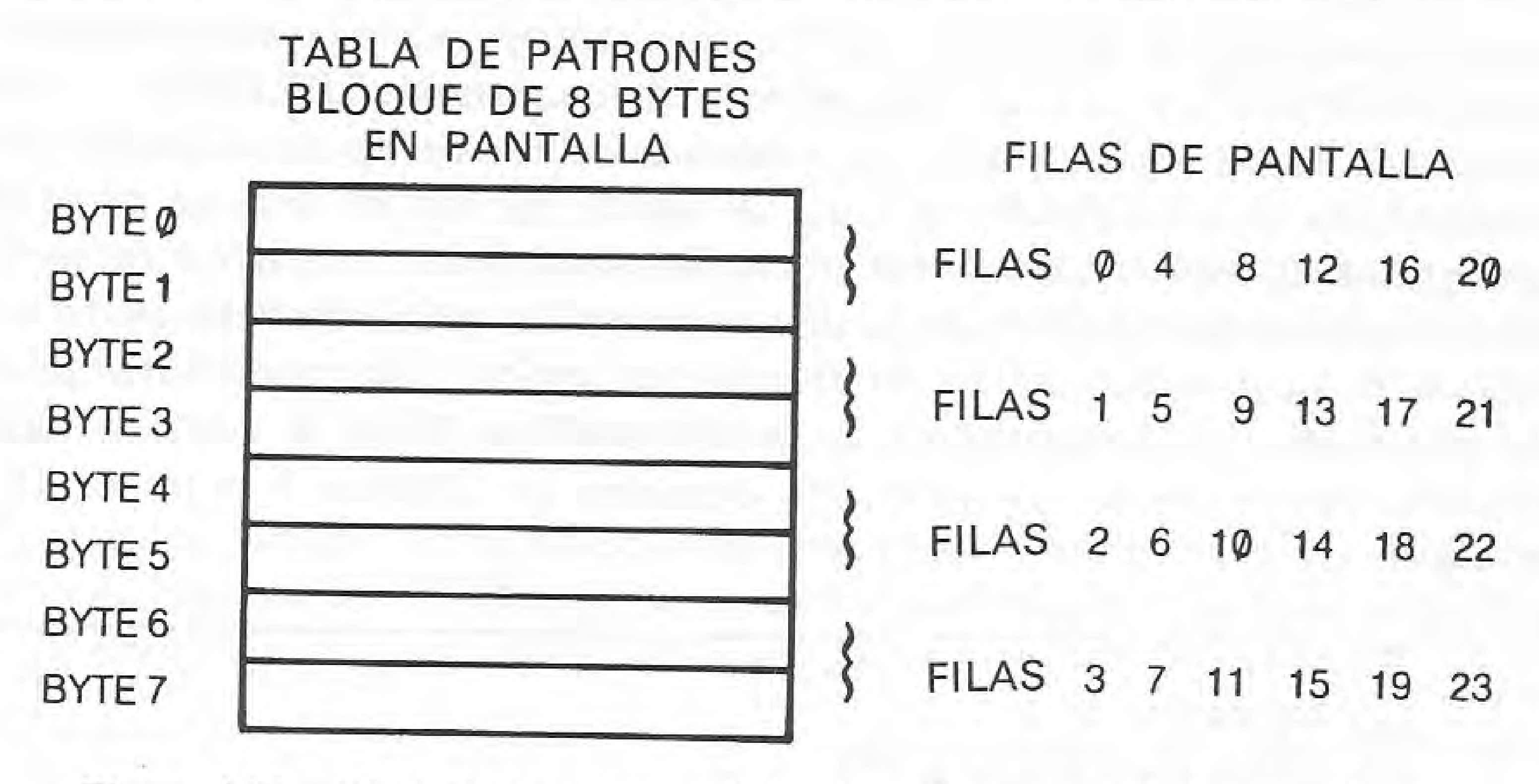


Figura 6.16.—*Mapa de los bloques de la tabla de patrones de la pantalla (2).*

Para clarificarlo más, imagine que la posición 0 de la tabla de nombres, que se corresponde con el primer carácter de la fila 0 de la pantalla, contiene el valor 2. Esto señala la posición (2 × 8), o 16 de la tabla de patrones. Aquí encontramos un

bloque de 8 bytes similar al dibujado en la figura 6.16. Los colores de los 4 *pixels* en esta posición de la pantalla estarían definidos por el byte n y el $n+1$, en la figura 13, debido a que la posición de la tabla de nombres está en la fila 0 de la pantalla. Si la posición 33 de la tabla de nombres contiene el valor 2, la imagen visualizada en la pantalla —que sería el primer carácter de la fila 1— estaría definida por los bytes $n+2$ y $n+3$ de la posición de la tabla de patrones. Este principio opera a todo lo largo de la totalidad de la tabla de nombres, causando por tanto que la imagen visualizada en la pantalla depende del valor contenido en la posición correspondiente de la tabla de nombres y de la fila de la pantalla a que corresponde esa determinada posición de esta tabla. El modo 3 ocupa 2.816 bytes de VRAM, sin contar las tablas de *sprites.*

Veamos ahora cómo el VDP almacena los datos referidos a los *sprites* en la memoria de video.

Tablas de "sprites"

Hay dos tablas en VRAM establecidas por el VDP en cada modo, que soportan *sprites.* Son la tabla de patrones de *sprites,* que contiene las definiciones de los *sprites* disponibles, y la tabla de atributos de *sprites,* que contiene información acerca del color y la posición de cada uno de ellos. La dirección de comienzo de cada una de estas tablas en un modo concreto es suministrada por el comando BASE apropiado, como ya hemos mencionado cuando estudiamos los modos de pantalla.

Tabla de patrones de "sprites"

Es un área de VRAM de 2.048 bytes, dividida funcionalmente en 256 bloques de 8 bytes. Esta tabla es accesible con el parámetro de número de patrón del comando PUT SPRITE, o con el parámetro *n* del comando SPRITE$(*n*). Contiene las definiciones de los *sprites* y por eso podemos definir un *sprite* alterando directamente los valores de la tabla de patrones de *sprites* en vez de utilizar SPRITE$(*n*).

El programa siguiente realiza esta operación, definiendo el patrón de *sprite* 0 y colocando los bytes que conforman la definición en los primeros 8 bytes de la tabla de patrones. Si quisiéramos definir en el plano de *sprite* 1 un modelo de *sprite* tendríamos que colocar la definición en las posiciones 8 a 15 de la tabla de patrones. La definición para el plano de *sprite* 256 ocuparía los últimos 8 bytes de la tabla de patrones.

```
10 SCREEN 1,1
20 comienzo=BASE(9)
30 FOR I=comienzo TO comienzo+7
40 VPOKE I,255
50 NEXT
60 PUT SPRITE 0,(100,100),1
```

Obviamente, si queremos definir *sprites* de 16 por 16 *pixels*, deben utilizarse 32 bytes para hacerlo, como ya hemos visto. Estos 32 bytes tendrían que colocarse en bloques de 8 en la VRAM. La necesidad de más datos para definir un *sprite* de 16 por 16 explica por qué están limitados a 64; ya que siempre hay la misma cantidad de VRAM disponible para almacenar las definiciones de *sprites*. Es posible cambiar la dirección de comienzo de la tabla de patrones de *sprites* alterando el contenido del registro 6 del VDP.

Tabla de atributos de "sprites"

Existe una tabla de atributos de *sprites* para cada plano de *sprites;* como hay 32 planos, hay 32 tablas, de 4 bytes de largo. Esto da una longitud total de 128 bytes para esta tabla. Cada una se organiza según lo expuesto en la figura 6.17.

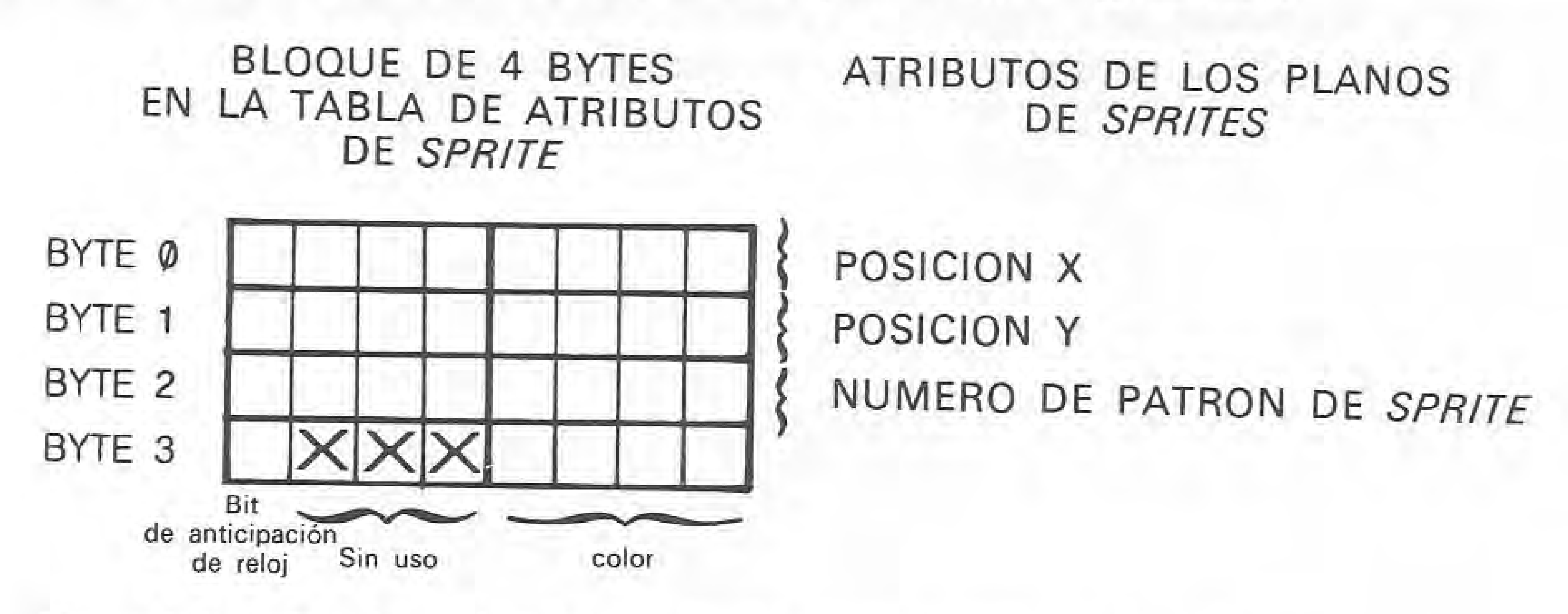


Figura 6.17.—*Funciones de los bytes en el bloque de atributos de* sprites.

El primer bloque de 4 bytes de la tabla de atributos se refiere al plano 0 de *sprites;* el segundo, al plano 1, etc. El número de patrón es el número de *sprite* que va a visualizarse en ese plano, y el color es el del *sprite* cuando aparezca en la pantalla. El primer byte de la tabla contiene la coordenada Y, y el segundo, la coordenada X del *sprite*. La única parte de la tabla que puede causar problemas para el principiante es el "bit de anticipación de reloj". Si ponemos este bit a 0, la esquina superior izquierda del *sprite* se coloca en la posición de pantalla X,Y. Sin embargo, si está a 1, la esquina superior izquierda del *sprite* se coloca en la posición X-32,Y.

El programa listado a continuación enseña cómo colocar un *sprite* en la pantalla mediante la instrucción VPOKE para acceder directamente a la tabla de atributos de *sprite* del plano del *sprite* a utilizar, en lugar del comando PUT SPRITE:

```
10 SCREEN 1,1
20 COMIENZO=BASE(9)
30 FOR I=COMIENZO TO COMIENZO+7
35 VPOKE 1,255
40 NEXT
```

```
50 COMIENZO=BASE(8)
60 VPOKE COMIENZO,100:REM posicion Y
70 VPOKE COMIENZO+1,100:REM posicion X
80 VPOKE COMIENZO+2,0:REM numero patron 0
90 VPOKE COMIENZO+3,1:REM color negro
```

Como puede observar, manejar *sprites* de esta manera es bastante engorroso; pero los principios apuntados aquí son de gran utilidad para el programador en código máquina del sistema MSX.

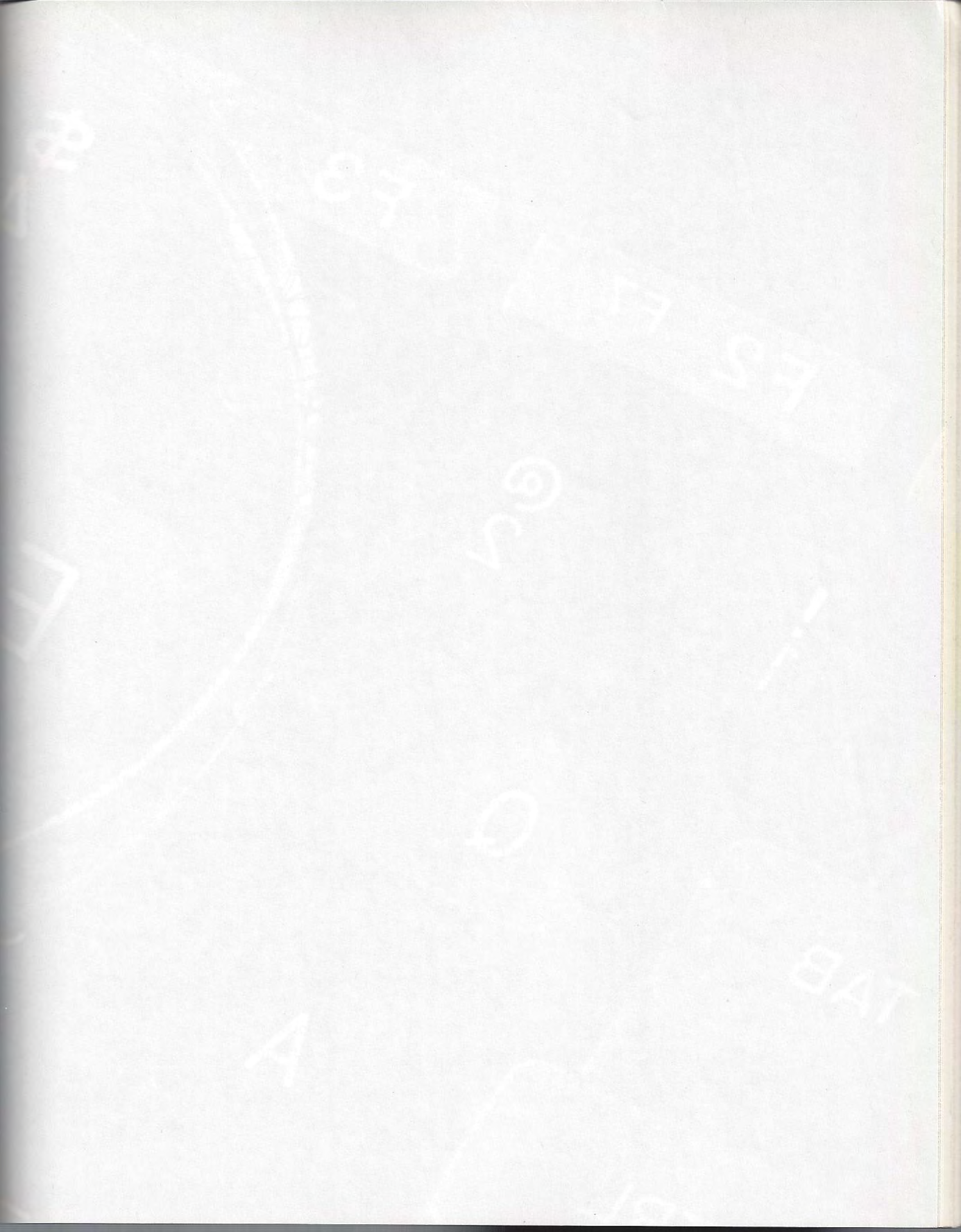
Se necesitan un total de 2.176 bytes para las dos tablas de *sprites*, suponiendo que se emplean los 256 patrones de *sprites*. Si no se hace así, pueden cambiarse los registros apropiados del VDP para producir tablas de atributos o de patrones alternativas. Recuerde no escribir sobre áreas de VRAM usadas.

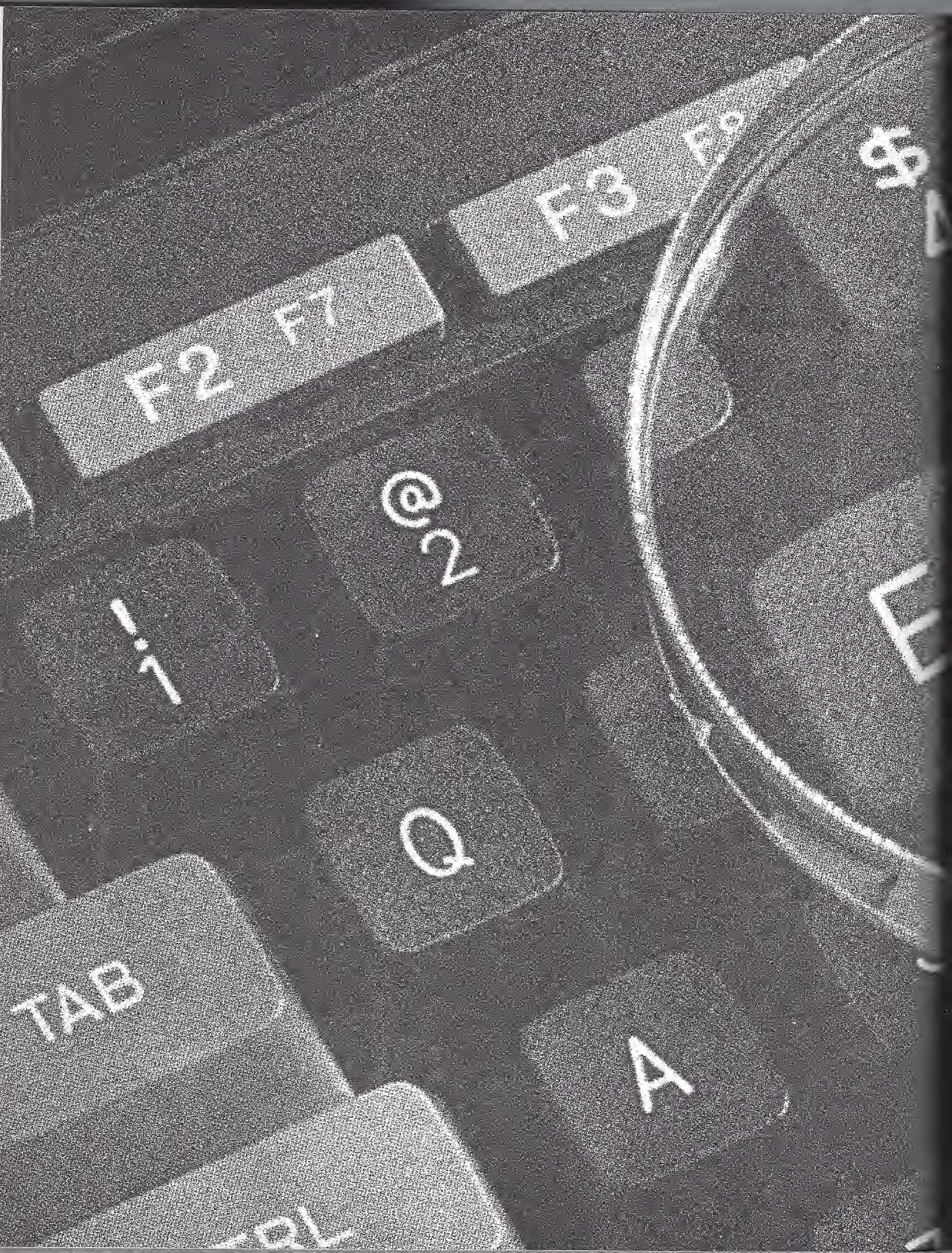
Con esto termina este capítulo acerca del VDP; nos lo encontraremos de nuevo, aunque de pasada, en el capítulo 11, cuando estudiemos cómo utilizar los registros del VDP y la VRAM basándonos en el código máquina.

NOTAS DEL CAPITULO

1.—SPRITE: La traducción literal es "duende" o "genio"; en los ordenadores personales se trata de un grupo de *pixels*, cuya forma y color puede definirlos el usuario, y que pueden desplazarse por la pantalla como un solo bloque. En los programas de juegos se utilizan con mucha frecuencia.

2.—MACROLENGUAJE: Es un lenguaje de nivel medio que en este caso se incluye como sublenguaje dentro del BASIC. El macrolenguaje gráfico y el macrolenguaje musical son los dos macrolenguajes que incluye el BASIC MSX, y que se estudian tanto en este capítulo como en el 8.




F2
F7
F3
$
@
2
!
1
Q
TAB
A



7 "Joysticks"

Uno de los aspectos interesantes del sistema MSX es que también da al usuario la posibilidad de conectar *joysticks* al ordenador; el BASIC MSX también provee de los comandos necesarios para manejarlos desde BASIC. También permite al programador simular *joysticks* usando las teclas del cursor y la barra de espaciado. Evidentemente esto es de gran utilidad para el progrqmador de juegos, especialmente si la máquina a utilizar tiene teclas del cursor grandes.

El comando usado para saber en qué dirección el usuario está manejando o simulando el *joystick* es STICK(*n*). *n* tiene un valor entre 0 y 2; 0 indica que la instrucción leerá las teclas del cursor, como si fueran un *joystick;* $n = 1$ hace referencia al *joystick* conectado al puerto 1, y $n = 2$, al conectado en el puerto 2.

El valor retornado por la función depende de la dirección en que el usuario desplaza el *joystick* que se está examinando. Si lo que utiliza son las teclas del cursor, el valor retornado indica la combinación de teclas pulsadas a la vez. La figura 7.1 muestra los valores retornados para las distintas combinaciones de teclas de cursor.

Si no se pulsa ninguna tecla al evaluar la función, o si el *joystick* no se mueve, la función da el valor 0.

La figura 7.2 expone los valores retornados según la dirección en que se desplaza el *joystick.* La dirección se da como punto del compás.

Si no se cambia la posición del *joystick,* el valor retornado es 0.

Los *joysticks* se conectan en los lugares marcados a tal efecto en el ordenador, a los que se accede como puerto 1 y puerto 2. La parte electrónica de los *joysticks* está manejada por el *chip* "generador de sonido programable", pero esto sólo tiene utilidad real para el programador en código máquina.

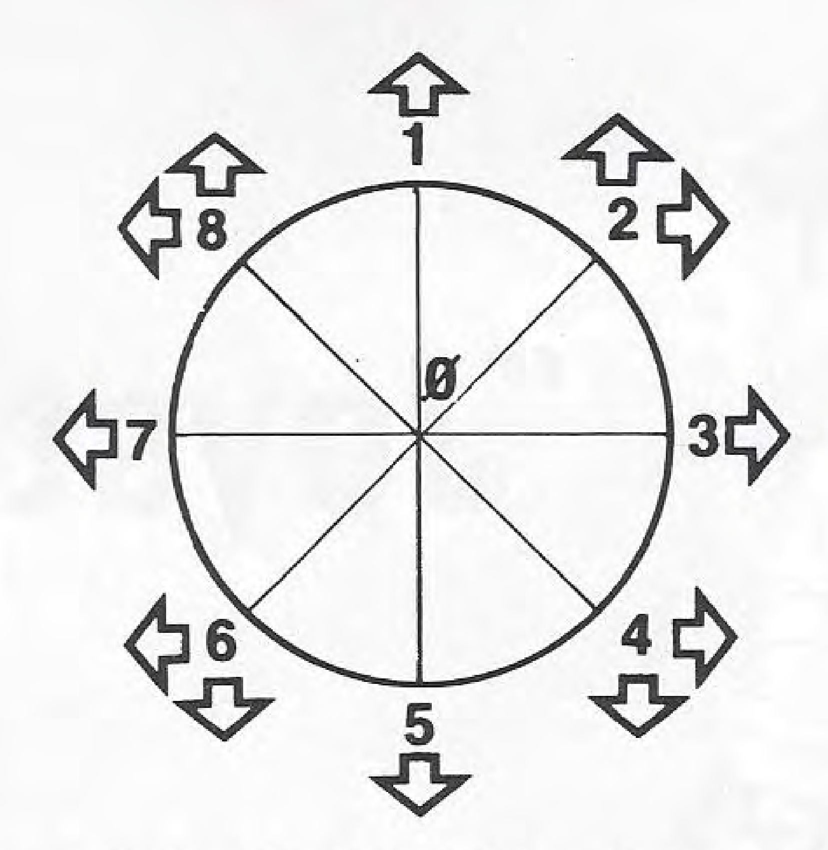


Figura 7.1.—*Lectura de dirección (1): de las combinaciones de las teclas del cursor.*

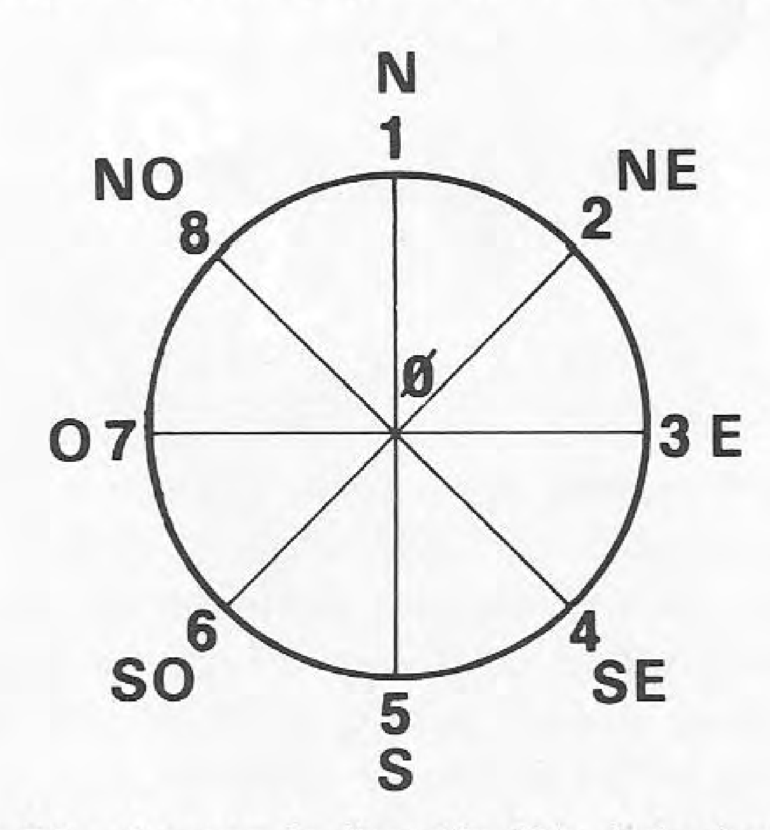


Figura 7.2.—*Lectura de dirección (2): de los* joysticks.

On strig[2]

Este comando nos permite hacer que el ordenador pase el control a una subrutina siempre que se oprima el botón de disparo del *joystick* o se pulse la barra de espaciado. La sintaxis completa es

```
ON STRIG(n) GOSUB numero de linea
```

donde *n* está entre 0 y 4, y el número de línea ha de ser válido y existente, al que se pasa el control cuando se produce lo indicado. Con los *joysticks* reales, $n = 1$ ó 3 provocará el salto a la subrutina cuando se oprima el botón de disparo del *joystick* 1, y $n = 2$ ó 4 provocará el salto si es oprimido el botón de disparo del *joystick* 2. Si $n = 0$, el salto tendrá lugar cuando se pulse la barra de espaciado.

STRIG(*n*) ON se usa para activar el salto a la rutina cuando se produce el

disparo. STRIG(*n*) OFF y STRIG(*n*) STOP tienen funciones análogas a los comandos KEY(*n*) OFF y KEY(*n*) STOP; si desea más detalles, consulte el capítulo 5.

Ahora veremos la forma de operar y cómo podemos programar el *chip* del generador de sonidos programables del ordenador.

NOTAS DEL CAPITULO

1.—PAD Y PADDLE: De hecho hay dos tipos de *joysticks:* analógicos y digitales. Con los analógicos tenemos conocimiento de la dirección y del grado de desplazamiento de la palanca del *joystick,* ya que usamos potenciómetros cuya resistencia varía al cambiar la posición de la palanca. Utilizando *joysticks* digitales sólo sabemos en qué dirección se empuja el mando, ya que en su construcción se emplean microinterruptores que se abren o se cierran según la posición del *joystick.* Todo lo expuesto en el capítulo 7 se refiere a *joysticks* digitales.

Para usar *joysticks* analógicos con BASIC hemos de tener en cuenta su construcción, porque para obtener una información completa tendremos que procesar los datos que nos proporciona la instrucción BASIC correspondiente. Esta instrucción es PDL(*n*), que proviene de PADDLE, y donde *n* es un número entre 1 y 12. En un *joystick* analógico, para cubrir todas las direcciones que puede tomar la palanca se utilizan dos potenciómetros montados perpendicularmente de tal manera que, mezclando adecuadamente las señales de cada uno de ellos, podemos conocer la posición exacta de la palanca en cualquier momento. Los ordenadores MSX no tratan los *joysticks* analógicos como un conjunto, sino que consideran los potenciómetros independientemente y se les suele llamar *paddle.* Cada puerto de *joystick* puede soportar hasta 6 *paddles,* correspondiendo al puerto 1 los valores de *n* 1, 3, 5, 7, 9 y 11, y al puerto 2 los valores de 2, 4, 6, 8, 10 y 12. Cuando se evalúa la función PDL(*n*) el sistema devuelve un número entre 0 y 255, que indica la posición del *paddle.* Por eso, por ejemplo, si utilizamos un *joystick* analógico para mover un *sprite* por la pantalla podríamos utilizar en el puerto 1 el *paddle* 1 y 3, y para obtener la posición en pantalla del *sprite* tendríamos:

```
coordenada X=PDL(1)           ; 0<X<255
coordenada Y=PDL(3)*192/255   ; 0<Y<192
```

Con respecto al *hardware,* los pines 1, 2, 3, 4, 6 y 7 de cada puerto de *joystick* corresponderían cada uno a un *paddle.*

Estos *paddles* se emplean también en los "ratones": idénticos a los *joysticks* analógicos, sólo que en vez de utilizarse una palanca móvil se usa una bola que gira, y en lugar de potenciómetros se utiliza un mecanismo un poco más complejo.

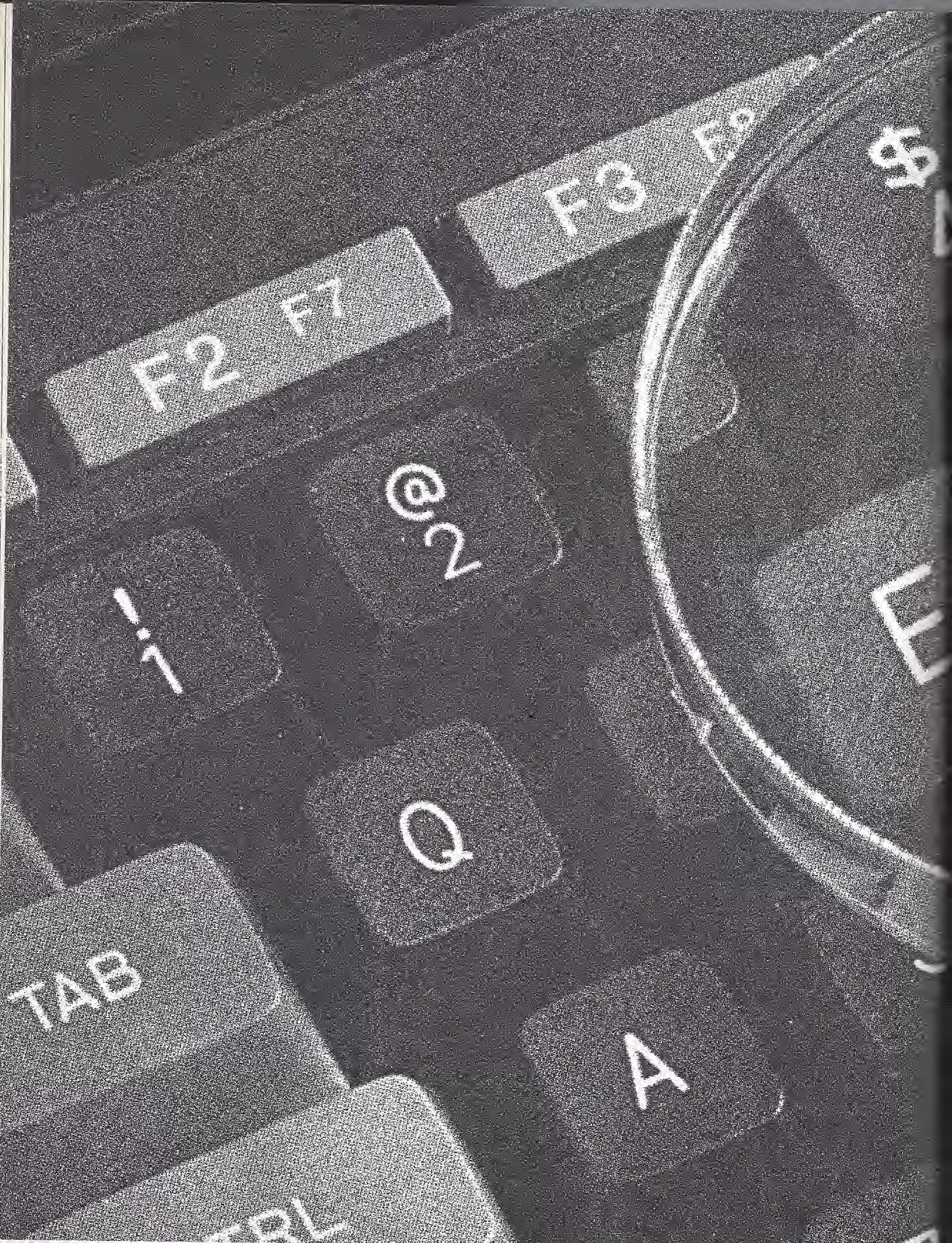
Otra instrucción BASIC referente al puerto del *joystick* es PAD(*n*) con *n* entre 0 y 7. Se utiliza cuando se conectan al ordenador tablas de gráficos (similares a los digitalizadores). Se puede conectar una de ellas a cada puerto del *joystick.* La tabla siguiente muestra los significados de la función según los valores de *n*:

0	−1 si se toca el *pad* 1 0 si no se toca el *pad* 1
1	coordenada X(0,255)
2	coordenada Y(0,255)
3	−1 si está pulsado el interruptor del *pad* 1 0 si no está pulsado el interruptor del *pad* 1

Con *n* entre 4 y 7 obtenemos resultados análogos para el *pad* 2.

2.—ON STRIG: Existen en el mercado *joysticks* con dos botones de disparo independientes. Los ordenadores MSX pueden operar con ellos, y por eso en las instrucciones STRIG(*n*) u ON STRIG(*n*) GOSUB, *n* puede tomar valores entre 0 y 4. Refiriéndose 0 a la barra de espaciado, 1 al botón 1 del *joystick* 1, 2 al 1 del *joystick* 2, 3 al 2 del 1 y 4 al 2 del 2.

Por otra parte, se puede utilizar STRIG(*n*) de manera similar a como lo hacíamos con STICK(*n*). Si al evaluar la función STRIG(*n*) obtenemos el valor −1, el botón está pulsado; en caso contrario toma el valor 0.


F2 F7
F3
@
2
!
1
Q
TAB
A



8 El sistema de sonido MSX

Todos los ordenadores MSX contienen un *chip* llamado generador de sonidos programable *(programmable sound generator)* o PSG. Este circuito integrado suele ser el *General Instruments* AY-3-8910, y su salida de sonido se envía al altavoz del televisor o a una clavija de audio externa. En este capítulo exploraremos los métodos de programación del PSG en BASIC, dando una introducción a la programación musical y de efectos sonoros.

El comando BASIC más simple que usa el PSG es BEEP. Produce el mismo sonido que se obtiene al pulsar CTRL-G. La sentencia

```
PRINT CHR$(7)
```

producirá también el mismo sonido que el comando BEEP. El otro sonido que nuestros ordenadores MSX pueden hacer en cualquier momento es el "*click* de tecla" *(key click)*, el ruido que hace el ordenador siempre que se pulsa una tecla. Lo único que podemos hacer con este "click" es activarlo o desactivarlo. La instrucción

```
SCREEN modo,tipo de sprite,0
```

desactiva el "click" de tecla, y la orden

```
SCREEN modo,tipo de sprite,1
```

lo vuelve a activar. No obstante, estos ruidos no son precisamente musciales. Pero el comando PLAY completa el conjunto de instrucciones de sonido del BASIC MSX. PLAY, de manera similar a la sentencia DRAW, implementa el macrolenguaje musical.

Esto nos permite componer música en el ordenador MSX con gran facilidad, como veremos a continuación:

PLAY

La sintaxis completa del comando PLAY es

```
PLAY cadena para el canal 1,cadena para el canal 2,
  cadena para el canal 3
```

Las cadenas pueden ser constantes o variables, como en el caso del comando DRAW. Las cadenas contienen letras, números y algunos caracteres no alfanuméricos (caracteres especiales). Las cadenas para el canal 2 y 3 son opcionales. El PSG puede tocar tres notas a la vez, teniendo cada nota tono y volumen diferentes. El primer grupo de caracteres que podemos poner en la cadena de una sentencia PLAY son las letras de la A a la G. Representando cada una una nota musical. El comando

```
PLAY "A"
```

tocará la nota A en el canal 1. La octava a la que pertenece la nota es la seleccionada en ese momento. La instrucción

```
PLAY "ABCDEFG"
```

tocará las notas musicales una detrás de otra en el canal 1. Un carácter "#", "+" o "−" puede seguir a una nota. Los caracteres "#" y "+" indican que la nota es un sostenido, y el carácter "−" indica que es bemol. Sin embargo, el símbolo de sostenido o bemol solamente es aceptado dentro de una cadena de comandos si corresponde a una nota bemol o sostenida del teclado del piano. Así, con el comando siguiente sonarán una serie de notas.

```
PLAY "C#DD#E"
```

Una octava en el macrolenguaje musical comienza en la nota C para acabar en la siguienee C. Para cambiar de octava utilizamos el comando O. El número que seguirá a la letra O indica la octava requerida. Este número, el número de octava, debe estar entre 1 y 8, siendo 1 la octava más baja disponible y 8 la más alta. Cuando empezamos a usar el ordenador, la octava preestablecida es 4. Por tanto, la sentencia

```
PLAY "CDEFGABO5C"
```

tocará una escala empezando en C de una octava y terminando en C de la siguiente. Una vez se ha emitido un comando de octava, a partir de ese momento todas las notas siguientes sonarán en esa octava hasta que se emita otra orden de octava. Esto se cumple aunque los comandos PLAY usados para tocar las notas siguientes sean totalmente diferentes. Pueden tocarse notas con facilidad en los tres canales disponibles; las notas suenan simultáneamente, posibilitando a los aficionados a la música tocar acordes. Por ejemplo, la instrucción

```
PLAY "C","E","G"
```

tocará el acorde de C mayor, mientras el comando

```
PLAY "C","D#","G"
```

tocará el acorde de C menor. Poniendo notas apropiadas en las tres cadenas pueden tocarse otros acordes. No obstante, como éste no es un libro de música, el lector interesado puede buscarse otro sobre la teoría y práctica de tocar acordes.

Es posible utilizar números en las cadenas de PLAY, en vez de nombres de notas. Para ello se usa la letra N, seguida de un número entre 0 y 96. La C en la octava 4 tiene un valor numérico para el sistema de 36. El siguiente comando tocará una cadena de notas definida por números.

```
PLAY "N1N2N3N4"
```

Personalmente, este método no supone ninguna ventaja real sobre el uso de nombres de notas.

No todas las notas de una pieza musical tienen la misma longitud; por eso, El BASIC MSX nos proporciona un medio para cambiar el tiempo de duración de una nota. El comando L *n* dentro de una cadena PLAY fijará la longitud de las notas siguientes a un valor que depende de *n* hasta que sea emitido el siguiente comando L. Las notas siguientes tienen una longitud de 1/*n* veces su longitud normal. El valor de *n* tiene que estar comprendido en el rango de 1 a 64: un valor de 1 da la máxima longitud para la nota; 4 dará un cuarto de la nota, y 64 dará a la nota la duración de 1/64 de una nota completa. Un ejemplo de su utilización se expone seguidamente:

```
PLAY "ABCL16DEF"
```

De nuevo, como en la sentencia Octava, la duración de la nota será la misma hasta que se ejecute el siguiente comando L. Olvidarlo puede causarle algunos problemas cuando escriba programas; por eso debe tenerlo siempre en mente. Si sólo se requiere el cambio de la longitud de un par de notas, las que quiere acortar pueden seguirse por el número deseado sin el comando L. Por ejemplo, en la instrucción siguiente, la nota A sonará durante 1/16 de la longitud de las otras notas:

```
PLAY "L1CDEA16DEF"
```

El valor de *n* utilizado generalmente es 4, si no emitimos ningún comando L. Si queremos dar una pausa corta entre notas, empleamos el comando R. El parámetro de este comando está otra vez entre 1 y 64, y la pausa de longitud igual a una nota completa se consigue colocando un 1 tras la letra R. Así, con la orden siguiente tendremos una pausa antes de tocar las tres notas de la cadena:

```
PLAY "R1ABC"
```

Un valor de 4 tras la letra R daría una pausa de longitud igual a un cuarto de nota, y con uno de 64 la pausa sería de 1/64 de la longitud de una nota.

Ya hemos visto cómo podemos acortar el tiempo que dura una nota utilizando el comando L; también es posible extender el tiempo de duración de la nota, mediante el carácter "·". Un solo punto siguiendo a una nota en una cadena alargará la duración de la nota a una vez y media. También puede usarse el punto para alargar la duración de un silencio de manera similar. Los efectos de un punto son acumulables; por tanto, con el ejemplo siguiente la nota sonará dos veces y cuarto su duración normal.

```
PLAY "A.."
```

¿De dónde salen las dos veces y cuarto (9/4)? Bien, simplemente es el resultado de $3/2 \times 3/2 = 9/4 = 2 + 1/4$. Está claro que pueden conseguirse notas de una duración extremadamente larga, usando repetidamente el comando.

Puede fijarse la velocidad a la que el ordenador interpreta nuestra composición, fijando el tiempo de la pieza musical. El comando T *n*, donde *n* es un valor entre 32 y 255, cumple esta función. *n* especifica el número de cuartos de nota tocados en un minuto. El valor tomado por defecto es 120.

El volumen del sonido producido se fija con el comando V. El parámetro *n* que se pasa al comando V ha de estar entre 0 y 15, siendo inaudible un valor de 0, y el volumen máximo, 15. El volumen del sonido, si no hay ninguna orden V, es 8.

Envolventes

El sonido producido hasta ahora siempre ha tenido un volumen fijado por el comando V, y no ha habido posibilidad de variarlo mientras la nota sonaba; es decir, la nota tenía la misma amplitud durante todo el tiempo que estaba sonando. Si lo comparamos con un instrumento real, como un piano, encontramos que en el piano la nota comienza con un volumen alto que decae lentamente. La nota del piano está "modelada", y a esta forma en la que la nota está modelada se le llama envolvente. Con los ordenadores MSX podemos simular esto en menor grado, donde hay ocho formas de onda diferentes que podemos seleccionar. Para hacerlo se emplea, en el macrolenguaje musical, el comando S *n*, donde el parámetro *n* tiene un valor entre 0 y 15, y ciertos valores de *n* producen la misma envolvente que otros. La

figura 8.1 muestra las distintas formas de envolvente disponibles, y el valor de *n* necesario para conseguirlas.

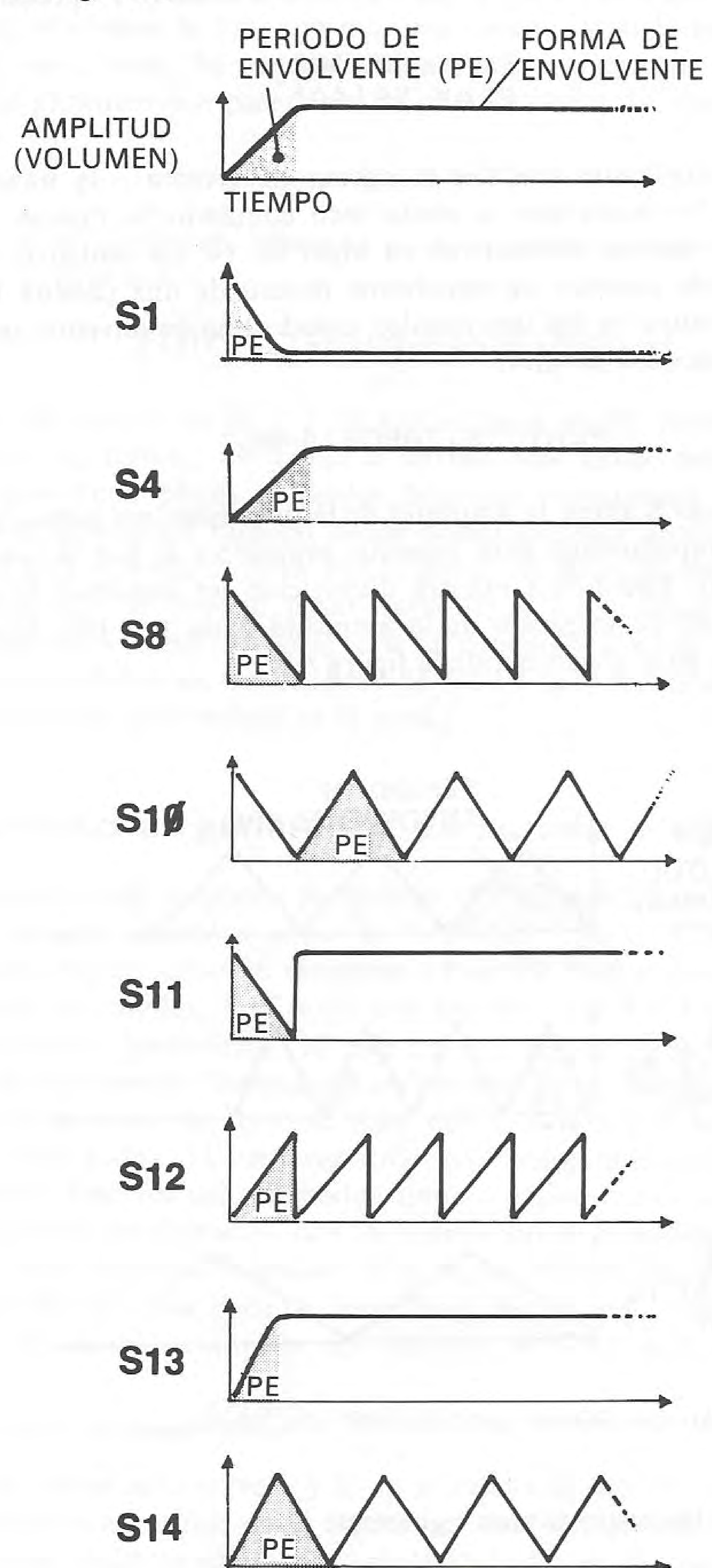


Figura 8.1.—*Modificación de envolvente (1): el macrocomando de sonido S.*

Para ver el efecto de estas envolventes, reinicialice el ordenador y pruebe los dos comandos listados a continuación. Escuche los dos sonidos y aprecie la diferencia.

```
PLAY "S13A"
PLAY "S14A"
```

Todas las notas siguientes tendrán la misma envolvente —la número 14 mostrada en la figura 8.1— hasta que se emita otro comando S. Pruebe las otras envolventes, colocando valores alternativos en lugar de 14. La sentencia siguiente demuestra que es posible cambiar de envolvente dentro de una cadena PLAY. También podemos tocar notas en los tres canales usando una envolvente, pero ésta debe ser la misma para todos los canales:

```
PLAY "S13ABCS14ABC"
```

Aunque el comando S altera la amplitud de la nota mientras suena, puede que no lo haga lo bastante rápidamente para nuestros propósitos o, por el contrario, puede ser demasiado rápido. Tenemos a nuestra disposición un comando llamado M que influye en la velocidad de variación de la amplitud dada por una envolvente concreta. Esto se expone en el diagrama de la figura 8.2.

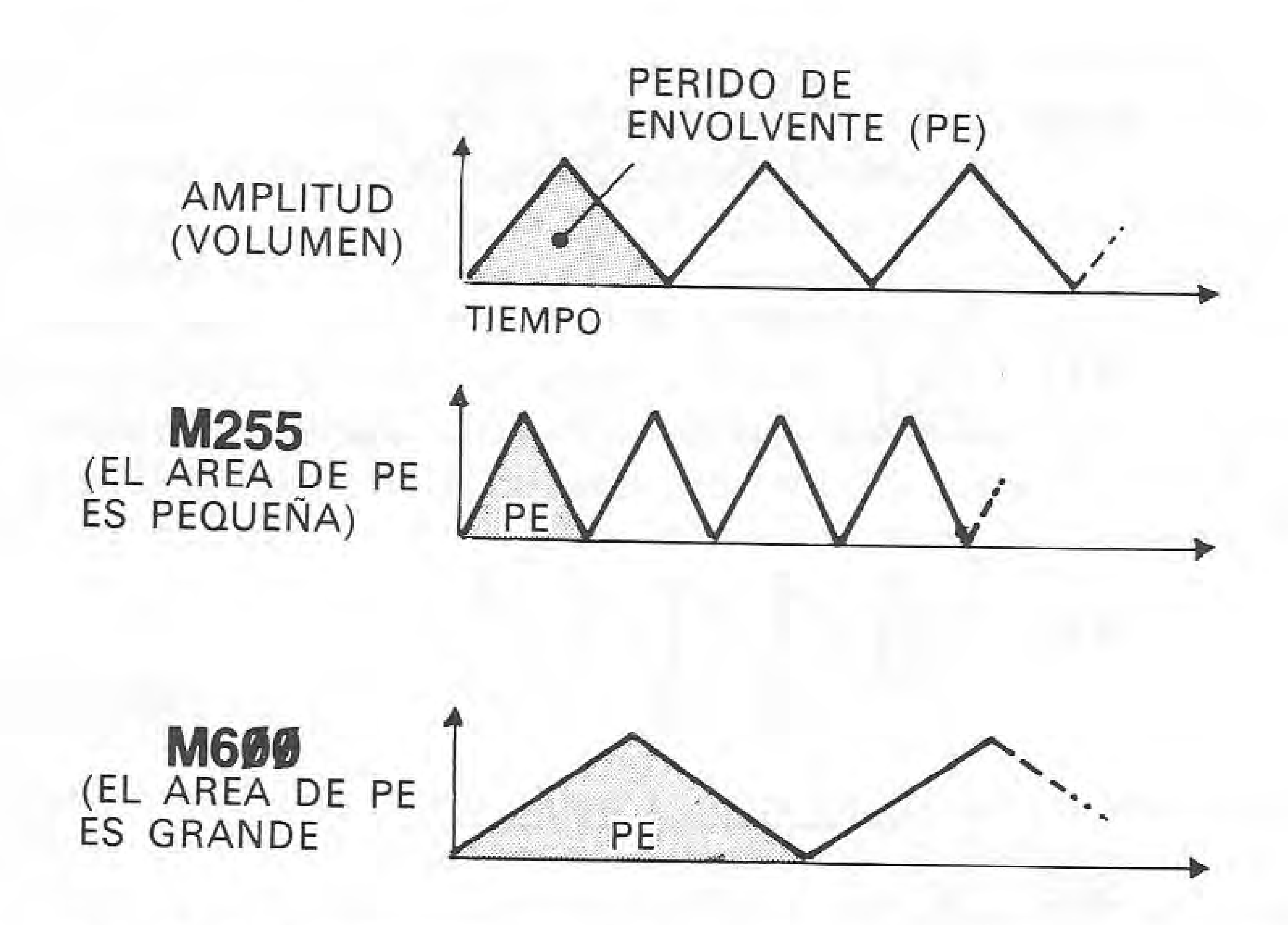


Figura 8.2.—*Modificación de envolvente (2): el macrocomando de sonido M.*

Para oírlo pruebe las instrucciones siguientes:

```
PLAY "S1M255A"
PLAY "S1M600A"
```

Observe que el segundo comando produce un "ping" de alta frecuencia opuesto al "zomp" del primer comando. El valor de *n* en el comando puede estar entre 0 y 65535. Esto determina la longitud del área mostrada en la figura 8.2 como "PE", o período de envolvente. Se pueden obtener efectos bastante interesantes variando el valor de los parámetros *n* pasados a los comandos S y M. Por ejemplo:

```
PLAY "L1S14M1000A"
```

da una "A" interesantemente alterada, y el comando siguiente produce un sonido similar al de un trombón:

```
PLAY "S14M100AR14AR14AR14A"
```

El uso de los comandos M y S de esta manera puede conducirnos a la generación de efectos de sonido, así como a darnos una gama más amplia de "voces" con las que interpretar piezas musicales. Mientras experimenta con los distintos comandos del macrolenguaje musical puede desear regresar al punto de donde partió; si pulsa CTRL-G reinicializará el PSG. Antes de dejar el macrolenguaje musical hemos de decir que se pueden pasar variables a las cadenas de la orden PLAY de manera similar a como se hacía con el comando DRAW. También la letra X tiene los mismos efectos en las cadenas de PLAY que en las de DRAW. (Consulte el capítulo 6 si desea profundizar en el tema.)

Acceso directo al generador de sonido programable

Veamos ahora cómo podemos programar el PSG modificando sus registros directamente, de manera similar a como lo hacíamos con el VDP. El generador de sonido programable contiene 16 registros, 14 de los cuales conciernen directamente a la generación de sonidos. Los otros dos registros del AY-3-8910 son de entrada/salida *(input/output)*, permitiendo al *chip* conectarse con otros componentes del sistema aparte del procesador central del ordenador. En el sistema MSX, uno de estos registros se utiliza como de entrada, para dar al ordenador acceso a los *joysticks*, si estuvieran conectados. El otro registro está configurado para salida y ayuda al control de varios aspectos del ordenador, que no explicaremos aquí.

Los 14 registros relacionados con la generación se modifican usando el comando SOUND, cuya función es similar a la de la instrucción VDP(*n*). No obstante, la orden SOUND no nos permite leer datos de los registros del PSG, como lo hacía VDP(*n*). La sintaxis completa del comando SOUND es la siguiente:

```
SOUND registro,valor
```

Registro es un número entre 0 y 13, y se refiere al registro del PSG a modificar. Valor es el número a escribir en el registro, y está comprendido entre 0 y 255. Sin embargo, algunos registros solamente aceptarán números con un valor bajo, apartándose de la regla general; esto se advertirá cuando los consideremos. Ahora vamos a ver la función de cada uno de los registros del generador de sonidos programable.

Registros 0 y 1

Los 8 bits del registro 0 y los 4 bits menos significativos del registro 1 conforman uno de 12 bits que controla la frecuencia del sonido producido por el canal 1. Este par de registros obviamente se modifica siempre que se emite un comando PLAY que cambie la frecuencia de la nota. Los 4 bits del registro 1 tienen mayor significación que los otros 8 bits, y aquí una variación tiene mayor efecto en la frecuencia del sonido producido que una variación similar en los contenidos del registro 0. Por esta razón, al registro 1 se le llama frecuentemente "registro de ajuste grueso de frecuencia", y al registro 0 se le llama "registro de ajuste fino de frecuencia". La frecuencia del tono generado en un canal dado del PSG depende del valor contenido en los registros de control de frecuencia correspondientes (registros 0 y 1 para el canal 1) y de la frecuencia de reloj aplicada. El valor más alto colocado en los registros de control de frecuencia genera el tono de frecuencia más bajo.

Registros 2 y 3

Ambos realizan un trabajo similar a los registros 0 y 1, respectivamente, pero para el canal 2. El registro 2 es el registro de ajuste fino de frecuencia, y el 3 es el registro de ajuste grueso de frecuencia del canal 2.

Registros 4 y 5

Ambos realizan la misma función que los registros 0 y 1, pero controlan la frecuencia del sonido producido en el canal 3. El registro 4 es el registro de ajuste fino de frecuencia, y el 5 es el registro del ajuste grueso de frecuencia.

Si, con esta información, acude a su ordenador y modifica los registros apropiados esperando escuchar algún sonido, se llevará una desilusión. La información para la frecuencia del sonido a generar por el canal 1 se escribiría de la siguiente manera:

```
SOUND 0,255:SOUND 1,0
```

pero no se han enviado datos al ordenador informándole sobre el volumen del sonido. Para generar un sonido puro necesitamos enviar al PSG la información sobre la frecuencia y amplitud del sonido. Se utilizan tres registros dentro del PSG para controlar la amplitud del sonido producido (cada registro controla uno de los tres canales).

Registro 8

Este registro controla la amplitud del sonido producido por el canal 1. El registro contendrá un valor entre 0 y 15, siendo 0 el volumen mínimo y 15 la amplitud o volumen máximos de la nota.

Registro 9

Igual que el registro 8, sólo que almacena los datos acerca del volumen del sonido producido por el canal 2.

Registro 10

Realiza el mismo trabajo que el registro 8, pero fija el valor del volumen para el canal 3.

Una vez se escribe un valor en uno de estos registros de control de amplitud, se generará un sonido en el canal apropiado. El programa que exponemos a continuación realiza este cometido:

```
10 SOUND 0,255:REM ajuste fino de
15 REM frecuencia del canal 1
20 SOUND 1,0:REM ajuste grueso de
25 REM frecuencia del canal 1
30 SOUND 8,10:REM amplitud del canal 1
40 TIME=0
50 IF TIME<50 THEN GOTO 50
60 SOUND 8,0:REM desactiva el tono
```

Observe que hemos fijado la amplitud del canal a 0 deliberadamente, para interrumpir el sonido. Si queremos generar sonidos en los tres canales simultáneamente, colocamos los valores apropiados en los registros de control de frecuencia correspondientes y terminamos el trabajo fijando los valores de los registros de control de amplitud, generando por tanto el sonido con el volumen elegido. Advierta que el uso de sentencias DATA agiliza el proceso de modificar los registros:

```
10 FOR I=1 TO 6
20 READ reg,v
30 SOUND reg,v
40 NEXT
50 DATA 0,255,1,0,2,200,3,0,4,150,5,0
```

Ejecute el programa anterior y después emita la línea siguiente como un comando directo:

```
SOUND 8,10:SOUND 9,10:SOUND 10,10
```

Ello fijará los tres registros de amplitud para que se produzca el sonido. Para interrumpirlo teclee CTRL-G o coloque los tres registros de amplitud a 0. Una vez hemos empezado a producir un sonido de esta manera podemos variar su fre-

cuencia alterando los registros de control de frecuencia de ese canal, como en el programa siguiente:

```
10 SOUND 1,3
20 SOUND 8,10
30 FOR I%=0 TO 255
40 SOUND 0,I%
50 NEXT
60 SOUND 8,0
```

La línea 10 fija el registro de ajuste grueso de frecuencia del canal 1, y la línea 20 activa la producción del sonido. Las líneas 30 a 50 modifican continuamente el contenido del registro de ajuste fino de frecuencia, resultando una variación continua de la frecuencia del tono generado. Una vez termine el bucle, la línea 60 interrumpe el sonido. Igual que cambiamos la frecuencia del tono de esta manera, también podemos alterar su amplitud mientras está sonando. En el programa siguiente lo vemos en acción:

```
10 SOUND 0,255:SOUND 1,0
15 REM ajuste de frecuencia
20 FOR I=0 TO 15
30 FOR J=0 TO 50:NEXT
35 REM bucle de memoria
40 SOUND 8,I
50 NEXT
60 SOUND 8,0
```

Produce un tono de volumen estrictamente creciente, hasta que se desconecta en la línea 60. Hasta ahora, la manera que hemos visto de interrumpir la producción de un sonido es hacer 0 su amplitud. Sin embargo, hay otro método, que es útil si sólo queremos interrumpir el sonido durante un momento, pero deseamos que continúe más tarde con el mismo volumen. Para ello se utiliza el registro 7, conocido como "registro mezclador".

Registro 7

Consideraremos este registro bit a bit, ya que cada uno de ellos controla un aspecto del PSG. La numeración de los bits se refiere al diagrama siguiente:

Bit 0. Gobierna el canal 1. Cuando está a 1, el tono de salida del canal 1 está totalmente inhibido, incluso si la amplitud fijada para el canal 1 no es 0. Poniendo este bit a 0 reactivaremos el tono de salida del canal 1. Tan pronto como este bit se pone a 0 se generará el sonido con la amplitud fijada en el registro 8, el registro de amplitud del canal 1. Obviamente, si el contenido del registro 8 es 0, el tono será inaudible.

Bit 1. Realiza la misma función que el bit 0, pero para el canal 2.

Bit 2. Realiza la misma función que el bit 0, pero para el canal 3.

Bit 3. Cuando está a 0, este bit activa la producción de ruido blanco en el canal 1, con la amplitud indicada en el registro de amplitud del canal 1. Cuando está a 1, se inhibe la salida de ruido blanco en el canal 1. Si este bit y el bit 0 de este registro están a 0, se producirán a la vez el ruido y el tono en este canal con la amplitud elegida y contenida en el registro 8. Enseguida daremos más detalles acerca del "ruido blanco".

Bit 4. Realiza un trabajo similar al del bit 3, pero para el canal 2.

Bit 5. Como el bit 3, pero actúa sobre el canal 3.

Bits 6 y 7. Controlan los registros de entrada y salida del PSG y no se utilizan para la producción de sonido. En la especificación del sistema MSX el bit 6 debe estar a 0 y el bit 7 a 1 para asegurar el correcto funcionamiento del ordenador.

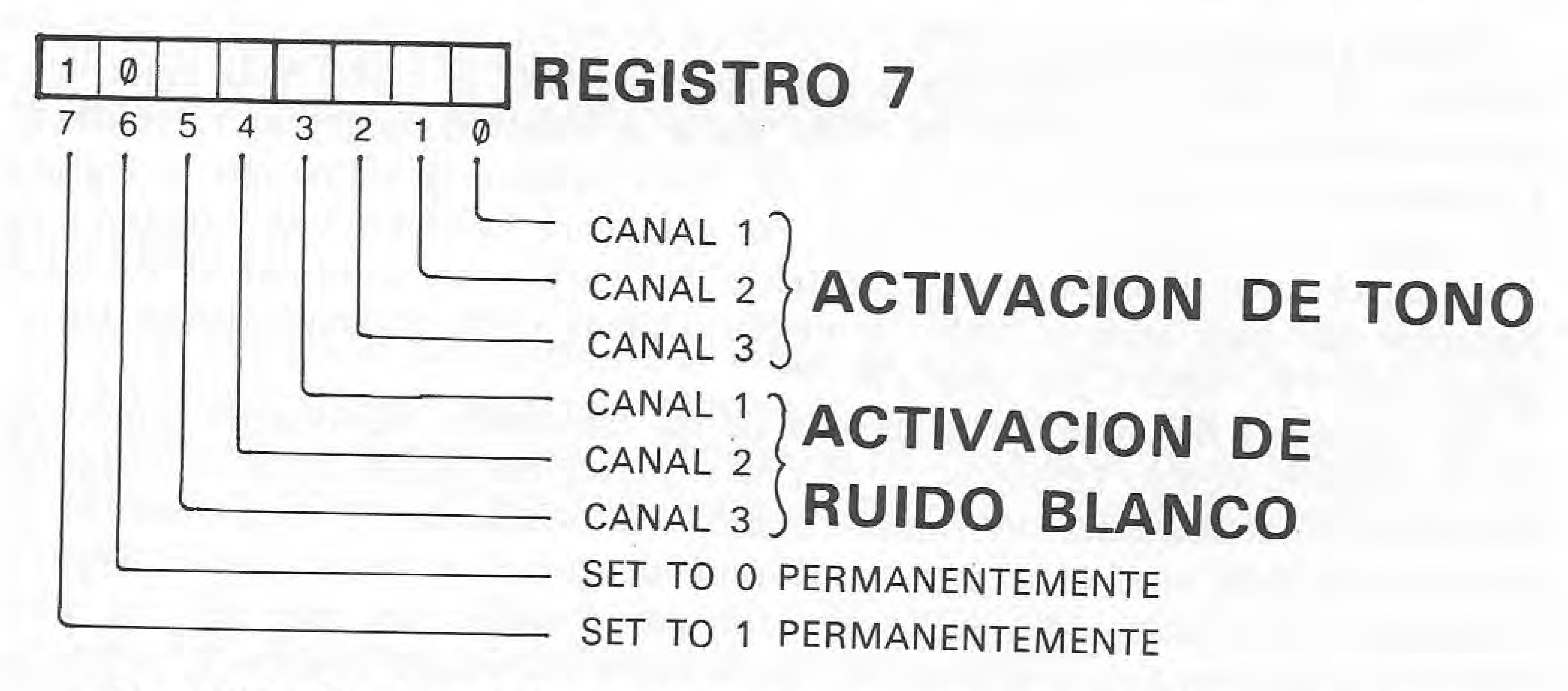


Figura 8.3.—*Funciones de los bits en el registro 7, generador de sonido programable.*

Quizá la mejor manera de modificar este registro es representar el valor a escribir como un número binario, ya que esto hace más fácil seleccionar el número de unos y ceros para activar las salidas de ruido y tono deseadas. Por ejemplo, el comando

```
SOUND 7,&B10111110
```

activará el canal 1 manteniendo inhibidas el resto de las salidas de sonido. El bit 6 está a 0 por la razón mencionada anteriormente. La orden activará el tono y el ruido del canal 1. Patrones similares de bits escritos en el registro activarán otros canales en forma similar. Por ejemplo, el comando

```
SOUND 7,&B10111000
```

activará el tono de los tres canales.

```
SOUND 7,&B10111000
```

Ruido

Ya hemos mencionado el ruido —o "ruido blanco", como se le conoce habitualmente— al estudiar el registro 7 del PSG. El ruido blanco se describe mejor en términos comunes como un ruido silbante; pruebe el programa siguiente para escucharlo:

```
10 SOUND 7,&B10110111
15 REM activa el ruido en el canal 1
20 SOUND 8,15
30 TIME=0
40 IF TIME<50 THEN GOTO 40
50 SOUND 8,0
```

Muchos sonidos naturales están compuestos de ruido; los típicos son la caída de la lluvia, el viento y las olas. En juegos podemos usar el ruido para imitar disparos, explosiones, etc. El ruido se define según su volumen, amplitud y frecuencia. La amplitud de un ruido producido en un cierto canal se gobierna con el registro de control de amplitud de ese canal. Por eso, para producir una explosión de ruido en el canal 1 usaríamos el registro 8 del PSG y así controlar el volumen. Recuerde que para generar ruido en un cierto canal, el bit correspondiente del registro 7 del PSG tendrá que estar a 0.

La frecuencia del ruido es la medida de las cantidades relativas de sonido de alta y baja frecuencia incluidos en el ruido. El registro 6 del PSG —el registro de control de la frecuencia de ruido— controla la frecuencia. El valor colocado en este registro debe estar entre 0 y 31; 31 provoca que el ruido producido tenga la frecuencia menor disponible, y 0 provocará que el sonido sea muy silbante. Para escuchar la diferencia de sonido según los distintos valores del registro 6, pruebe el programa siguiente:

```
10 SOUND 7,&B10110111
20 SOUND 8,6
30 FOR I=0 TO 30
40 SOUND 6,I
50 I$=INPUT$(1)
60 NEXT
70 SOUND 8,0
```

Ejecútelo; la presión sobre una tecla motivará que se coloque el siguiente valor en el registro y, por tanto, modificará la frecuencia del ruido generado. La siguiente rutina es un ejemplo de un efecto de sonido simple utilizando la disponibilidad del ruido: un disparo. La línea 20 fija el volumen máximo del ruido del canal 1. Las líneas 30 a 50 varían la calidad del sonido producido alterando los valores colocados en el registro de control de frecuencia de ruido y del registro de amplitud del canal 1. La línea 60 interrumpe la generación del sonido.

```
10 SOUND 7,&B10110111
20 SOUND 8,15
30 FOR I=31 TO 0 STEP -0.5
40 SOUND 6,I:SOUND 8,I/2
50 NEXT
60 SOUND 8,0
```

Una manera mucho más fácil de producir efectos de sonidos es usar la capacidad del PSG de producirlos con envolventes. Lo vimos operar con el comando S del macrolenguaje musical.

Envolventes

Hay tres registros implicados en el control de envolventes; son el 11, 12 y 13. Veámoslos por orden y estudiemos qué funciones realizan.

Registro 13

Este es el registro de control de la forma de envolvente, y puede contener valores entre 0 y 15. Las formas de envolvente obtenidas para los distintos valores del registro son las mismas que se obtuvieron para ese valor en el comando S del macrolenguaje musical. Observe la figura 8.1 para ver las distintas formas de envolvente.

Registros 11 y 12

Son los registros de control del período de envolvente y los del PSG modificados por el comando M del macrolenguaje musical. El registro 11 contiene los 8 bits menos significativos del parámetro de 16 bits que pasábamos al comando M, y el registro 12 contiene los 8 bists más significativos de este parámetro. Así, los comandos

```
SOUND 11,255:SOUND 12,255
```

y

```
PLAY "M65535"
```

realizan la misma función.

Por ello, podemos fijar la envolvente y el período que queramos, pero ¿cómo aplicarlo para producir un sonido completo? Hacemos esto fijando a 16 el registro de amplitud del canal en el que queremos producir el sonido con envolvente. Y para conseguir que el sonido producido en el canal 1 adopte la forma de la envolvente colocaremos en los registros 11, 12 y 13 los valores apropiados, y fijaremos a 16 el registro 8.

El programa siguiente produce un tono controlado por envolvente en el canal 1:

```
5 SOUND 7,&B10111110
20 SOUND 13,14
25 REM fija la forma de la envolvente
30 SOUND 11,255:REM activa el periodo
40 SOUND 12,0:REM de envolvente
50 SOUND 0,255:REM activa el tono
60 SOUND 1,0
70 SOUND 8,16:REM toca la nota
```

Este sonido continuará hasta que coloque el contenido del registro 8 a 0, o hasta que teclee CTRL-G. La envolvente definida así puede usarse para controlar la amplitud de un ruido igual que hacemos con el tono. Esto se puede demostrar cambiando la línea 5 por

```
SOUND 7,&B10110111
```

activando por tanto el ruido del canal 1 en lugar del sonido. Desde luego, puede activar ambos a la vez si lo desea.

Como habrá observado, presionando CTRL-G reinicializa los registros del PSG a los valores originales que tenían antes de que empezase a programarlos. Esto puede serle de gran utilidad ¡si no sabe cómo solventar un problema tras haber alterado los registros!

Con lo expuesto completamos la supervisión de la manera de programar el PSG desde el BASIC. Este circuito integrado necesita mucha experimentación para sacarle provecho; una buena sugerencia, por tanto, es experimentar algunos efectos propios. Para empezar, pruebe el siguiente programa:

```
10 FOR I%=0 TO 13
20 READ N
40 SOUND I%,N
50 NEXT
60 DATA datos apropiados
```

El programa lee valores de una sentencia DATA colocándolos en los registros del PSG apropiados. Para cambiar el sonido producido, simplemente altere las sentencias DATA y ejecútelo de nuevo. El primer término de la sentencia DATA es el valor del registro 0 del PSG; el segundo, el del registro 1, etc.

Disparo. Suena en los canales 1, 2 y 3:

```
DATA 0,0,0,0,0,0,17,7,16,16,16,1,5,1
```

Explosión. Suena en los canales 1, 2 y 3:

```
DATA 0,0,0,0,0,0,31,7,16,16,16,0,60,0
```

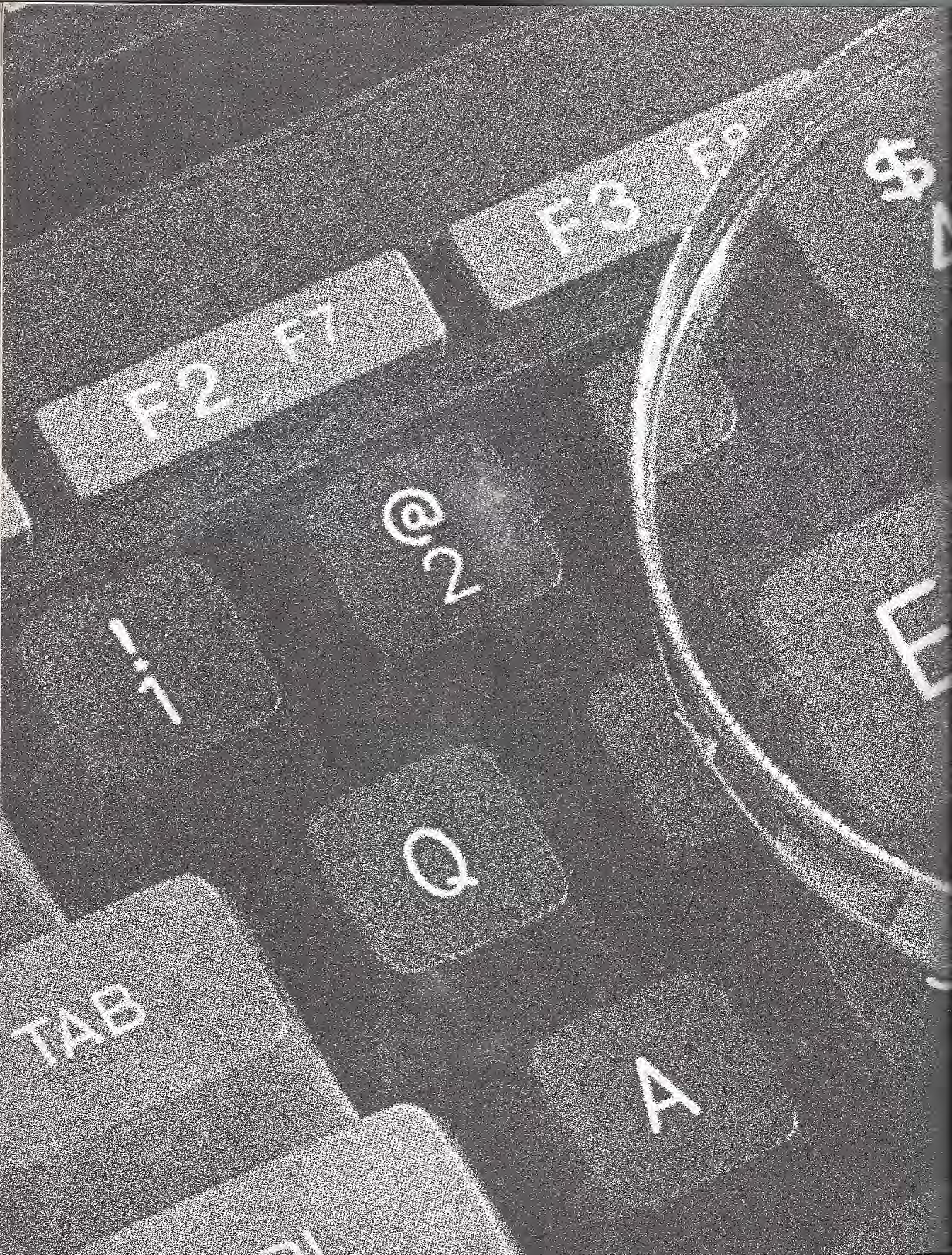
Boing. Canal 3:

```
DATA 0,0,0,0,0,0,9,&B10110111,16,0,0,191,5,1
```

Rayo Laser. Canal 3:

```
DATA 128,1,0,0,0,0,1,54,16,0,0,251,10,15
```

Otros efectos de sonido pueden sintetizarse fácilmente utilizando bucles FOR ... NEXT para modificar los contenidos de los registros del PSG, como ya hemos visto. Si usted tiene aficiones musicales, es posible sintetizar efectos que suenen como baterías y otros instrumentos de percusión. No obstante, la única manera de dominar realmente el PSG es a través de la experimentación. Le deseo suerte.


F2 F7
F3
@
2
!
1
Q
TAB
A
$
E



9 El interfaz periférico programable

PPI es el interfaz periférico programable, y sus siglas corresponden a *Programmable Peripheral Interface*. Se estudió someramente en el capítulo 1, viendo su función principal; en este capítulo lo examinaremos con más detalle. La primera advertencia es que, aquí, el estándar MSX falla ligeramente; el PPI se puede modificar mediante comandos llamados INP y OUT, que no escriben registros sino direcciones particulares, y por eso necesitamos saber las direcciones de memoria del PPI antes de poder acceder a ellas. El problema es que algunos fabricantes pueden variar las posiciones de memoria del PPI. Por tanto, las direcciones dadas para el PPI en este capítulo serán las del ordenador Sony HB-55 MSX[1]. No obstante, es posible que también funcionen en su ordenador correctamente. El problema de incompatibilidad entre máquinas será mínimo, sin embargo, usando rutinas del sistema operativo para acceder al PPI. No obstante, esto sobrepasa el alcance de este libro.

La mejor manera de empezar este capítulo es presentando los comandos BASIC utilizados para acceder al PPI. Son OUT e INP.

La sintaxis completa del comando OUT es

```
OUT numero de puerto,valor
```

Número de puerto es un valor dentro del rango de 0 a 255. Este es un buen punto para introducir el concepto de puertos de entrada/salida *(input/output)* en el sistema MSX. La unidad central de proceso Z-80 puede, como ya hemos mencionado, direccionar 65.535 bytes de memoria. Pero también puede acceder a 256 posiciones

más, llamadas DIRECCIONES IN/OUT. Están totalmente separadas de la RAM o ROM normales, y en el sistema MSX proveen del medio por el que la CPU se comunica con el procesador de display de video y el generador de sonido programable. Algunas de estas direcciones se emplean también para permitir a la CPU comunicarse con el interface periférico programable. Mientras que para acceder a las posiciones de RAM y ROM normales utilizamos PEEK y POKE, para las direcciones de IN/OUT hemos de usar los comandos INP y OUT. La sintaxis completa de la orden INP es

```
INP (numero de puerto)
```

donde número de puerto tiene un valor entre 0 y 255. Por eso, para leer datos de la dirección IN/OUT &HA8, e imprimirla en la pantalla, emitiríamos este comando:

```
PRINT INP(&HA8)
```

Para enviar un valor a la dirección IN/OUT &HA8 emitiríamos la orden

```
OUT &HA8,23
```

donde 23 es el valor a escribir en la dirección.

De cualquier modo, sigamos con el PPI. Contiene cuatro registros, tres de los cuales son de entrada/salida *(input/output)*, que interactúan (actúan de interfaces) con el teclado, el cassette, o la electrónica de selección de *slots* (esto último lo veremos en el capítulo 10). El cuarto registro se llama REGISTRO DE SELECCION DE MODO, y controla si los tres registros IN/OUT están configurados para entrada, salida o ambas cosas. Veremos este registro ahora. Lo primero y principal que podemos decir acerca de él es ¡no lo toque! Si altera su valor puede desconectar el teclado y la memoria ocasionalmente. Si quiere jugar con este *chip*, una buena sugerencia es que se limite a leer o escribir en los tres puertos de IN/OUT. Sin embargo, si todavía está interesado, le aconsejaría que consiguiese la "hoja de datos" del circuito integrado 8255 PPI. Este registro está configurado de manera que dos de los otros son para salida y uno para entrada. La organización de los tres registros en el espacio de las direcciones de I/O se expone a continuación. Las direcciones dadas son las establecidas en la especificación del estándar MSX.

&HA8 Registro A:OUTPUT (salida)
A9 Registro B:INPUT (entrada)
AA Registro C:OUTPUT (salida)
AB Registro de selección de modo

Direcciones del PPI

Recuerde, sin embargo, que éstas pueden no ser las direcciones correctas de su sistema en particular: el fabricante podría haber decidido alterarlas un poco.

Bien, ¿qué se puede hacer con los registros?

Registro A. Es un registro de salida, y se utiliza para controlar la localización de la memoria del sistema MSX. En el capítulo 10 veremos más detalles. Por el momento es suficiente con decir que no es aconsejable alterar su contenido a menos de que esté seguro de lo que está haciendo.

Registro B. Es un registro de entrada, y retorna, cuando lo leemos, un valor relacionado con el teclado. Este registro solamente nos proporciona una respuesta coherente si lo consideramos junto con los 4 bits menos significativos del registro C.

Registro C. Es un registro de salida, y su función principal es ayudar a la lectura del teclado. Los bits de 0 a 3 del registro forman lo que se llama la "señal de exploración del teclado" *(keyboard scan signal)*. El sistema MSX fija estos bits en un valor concreto, y después lee el valor del registro B. Para cada modelo binario en la salida del registro C pueden detectarse ocho teclas. Cada una de estas teclas, cuando es pulsada, hace que un bit del registro B se ponga a 0. Esto cambia el valor leído del registro B, y por eso nos permite leer el teclado directamente. La figura 9.1, el mapa del teclado MSX, expone cómo se realiza esta función.

Por ejemplo, un valor de &B0100 en los 4 bits menos significativos del registro C causará que puedan ser detectadas las teclas R,Q,P,O,N,M,L y K si se pulsa cualquiera de ellas o varias a la vez. Si se pulsa la tecla R, el bit 7 del registro B se pone a 0; por tanto, si leemos este registro obtendremos un valor de 127. Como podemos ver en la tabla, este método de lectura del teclado nos proporciona un medio de detectar teclas tales como SHIFT y TAB, que normalmente no lo son directamente. La siguiente rutina muestra los principios de la lectura del teclado de esta manera:

```
10 OUT &HAA,&B00000111
20 PRINT INP(&HA9)
30 GOTO 10
```

Ejecute el programa y pulse teclas tales como SELECT, ESC y BS. Obviamente, este programa en concreto solamente funcionará si el fabricante de la máquina ha respetado la especificación original.

Bit 4. Es la señal de control del cassette, y gobierna el relé del control remoto del cassette. Cuando este bit está a 0, el relé está cerrado, permitiendo por tanto operar con el cassette. Cuando está a 1, el relé está abierto.

Bit 5. Este bit está relacionado con la transferencia de datos a la cinta de cassette.

Bit 6. Este controla el estado de la luz de bloqueo de algunas teclas *(caps lock)*; es decir, el piloto que indica si está activo o no el *caps lock*. Cuando este bit está a 0, el piloto está encendido, y cuando está a 1, apagado. Este bit no influye en el estado de la función *caps lock*, sino sólo en el del piloto.

Bit 7. Colocando este bit a 1 y después a 0, oiremos un "click" producido por el sistema de sonido del ordenador. En el programa lo vemos en acción. Estará de acuerdo conmigo, supongo, en que ¡la naturaleza del sonido dista mucho de ser musical!

```
10 OUT &HAA,&B00000000
20 OUT &HAA,&B10000000
30 GOTO 10
```

Hemos cubierto todas las funciones del PPI exceptuando el registro A. Examinaremos las funciones de este registro cuando estudiemos en su totalidad la distribución de la memoria en el sistema MSX.

REGISTRO B* Modelos de bits leídos en el registro B								REGISTRO C** Modelos de bits escritos en el registro C
7	6	5	4	3	2	1	Ø	3 2 1 Ø
7	6	5	4	3	2	1	Ø	ØØØØ
;	[	@	¥	∧	—	9	8	ØØØ1
B	A	—	/	•	,	]	:	ØØ1Ø
J	I	H	G	F	E	D	C	ØØ11
R	Q	P	O	N	M	L	K	Ø1ØØ
Z	Y	X	W	V	U	T	S	Ø1Ø1
F3	F2	F1		CAPS	GRAPH	CTRL	SHIFT	Ø11Ø
RE-TURN	SELECT	BS	STOP	TAB	ESC	F5	F4	Ø111
→	↓	↑	←	DEL	INS	HOME CLS	SPACE	1ØØØ

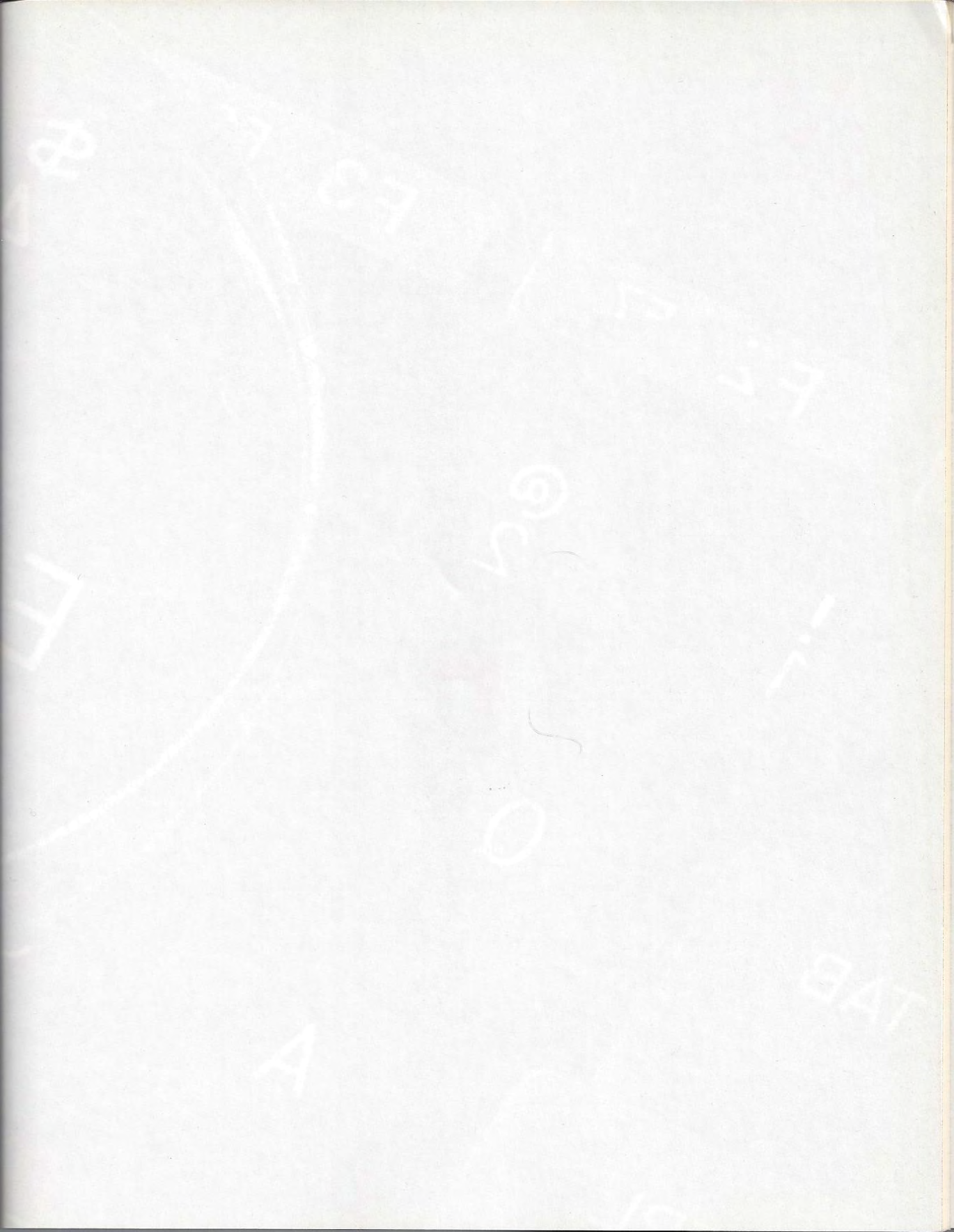
(*) Valor leído del registro B por un INP de 0A9H; todos los bits están a 1 excepto los que corresponden a una tecla pulsada, que están a 0.

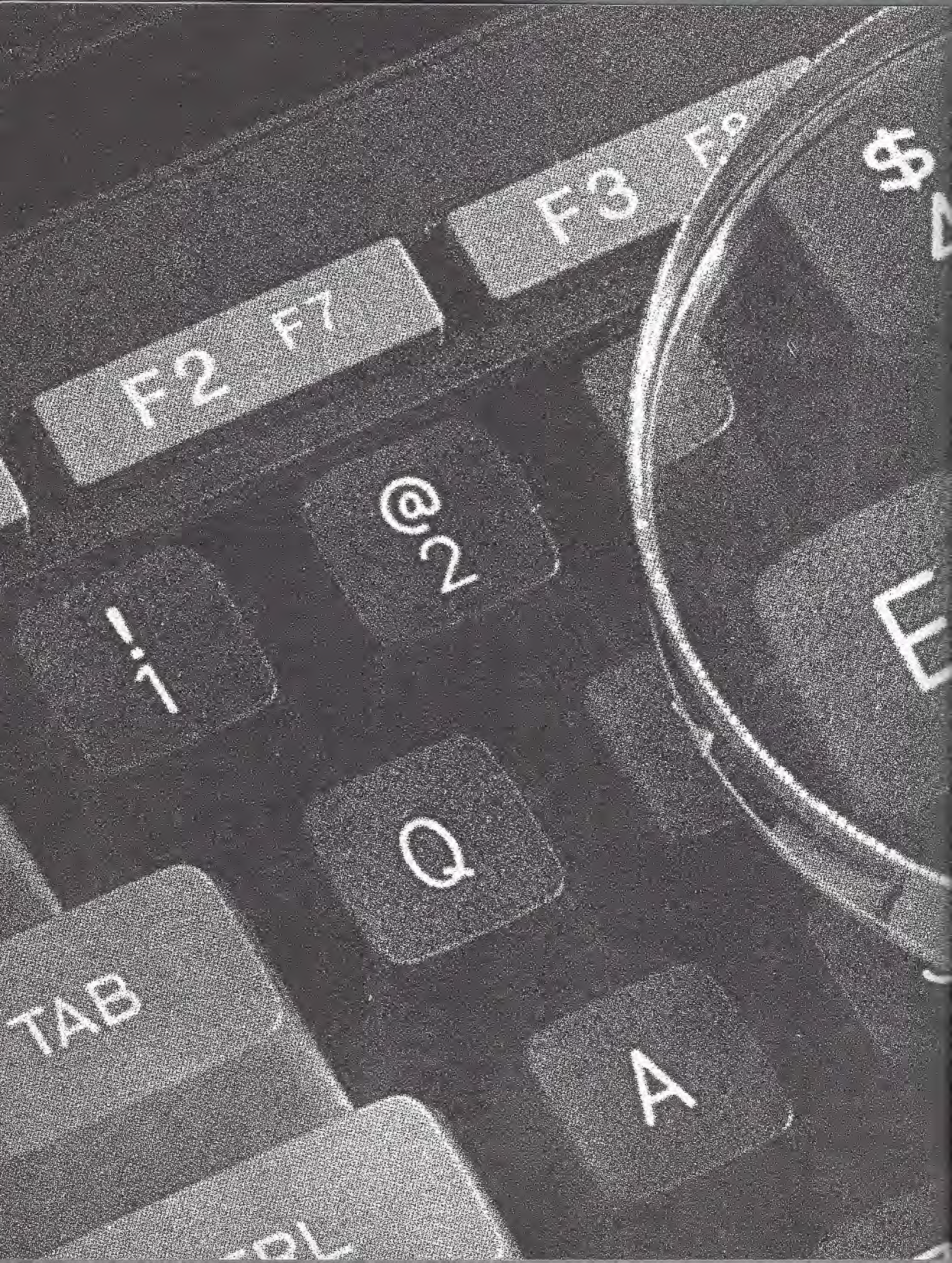
(**) El *nybble* menos significativo se escribe en el registro C con un OUT a 0AAH.

Figura 9.1.—*La tabla del teclado MSX.*

NOTA DEL CAPITULO

1.—Se corrobora que estas direcciones son correctas para el ordenador Spectravideo 728 MSX.




F2 F7
F3
@
2
!
1
Q
TAB
A
$
E



10 El mapa de memoria MSX

El mapa de memoria de un sistema es simplemente una descripción de cómo están organizados la memoria del ordenador y los *chips* de entrada/salida *(input/output)*. Antes de ir más lejos, le aconsejaría que consultase el apéndice sobre sistemas de numeración, si no lo ha hecho ya. Ya hemos expuesto someramente las reglas que rigen la RAM y la ROM, cuando en el capítulo 1 dimos una visión general del sistema. En este capítulo veremos con mucho más detalle la organización de la memoria en el sistema MSX, así como los *chips* de entrada/salida a los que podemos tener acceso.

La ROM MSX siempre empieza en la posición 0 de memoria y se extiende hasta la &H7FFF. Las posiciones de memoria entre &H8000 y &HFFFF están disponibles para RAM si se desea. El mapa de memoria para la especificación mínima MSX es el de la figura 10.1. La cantidad mínima de RAM que el sistema MSX necesita es 8K, pero sería muy limitado. Esta está colocada a partir de la dirección &HFFFF. Aunque la RAM del sistema mínimo es de 8K, sólo podemos añadir RAM en bloques de 16K. Si añadimos RAM al sistema mínimo, los 16K sumados nos ocuparían las posiciones del mapa de memoria desde la &HFFFF hasta la &HC000, un total de 16.384 posiciones. Esto puede causar cierta sorpresa si esperaba ser capaz de añadir 16K de memoria a los 8K del sistema y tener al final 24K. Como seguramente habrá comprobado, no es así; los 8K que ya estaban presentes se "tapan" con los 16K añadidos, debido al hecho de que el sistema MSX maneja su memoria en bloques de 16K, llamados PAGINAS. Las direcciones mostradas en la figura 10.1

se llaman LIMITES DE PAGINA. Por tanto, un sistema MSX con 16K de RAM se dice que tiene una página de RAM, que se extiende desde &HC000 hasta &HFFFF, y dos páginas de ROM ocupando el espacio desde &H0000 hasta &H7FFF.

Hasta ahora no hay nada peculiar acerca de la organización de la memoria en el sistema MSX. Lo que hace al sistema MSX diferente de otros sistemas, atendiendo a la organización de memoria, es el concepto de *slot*, que efectivamente nos permite añadir más memoria al sistema, que podría tener hasta 32K de RAM[1] y 32K de ROM. Un sistema como éste tiene un mapa de memoria completo; la CPU Z-80 solamente puede manejar 64K de memoria a la vez. El *slot* es un bloque de 65.535 (64K) posiciones que pueden ocuparse por RAM o ROM. Todos los ordenadores MSX deben tener al menos dos *slots*, pero pueden tener hasta cuatro. Uno de estos *slots*, el que contiene la ROM del BASIC MSX y la RAM normal de la máquina, se llama *SLOT* DEL SISTEMA *(system slot)* o *slot* 0. El segundo *slot* que poseen todos los ordenadores MSX se llama *slot* 1 o *SLOT* DEL CARTUCHO *(cartridge slot)*. Esto proviene de la conexión para cartucho de su máquina, en la que puede insertar RAM adicional o cartuchos de programas en ROM *(firmware)*. La manera en que el ordenador usa los *slots* es bastante complicada; pero, en términos simples, el ordenador puede, si lo necesita, hacer uso de páginas de diferentes *slots* para configurar su mapa de memoria. La figura 10.2 nos muestra un ejemplo.

El *slot* 0 contiene la ROM MSX y una página de RAM, comenzando en la posición &HC000 y acabando en la &HFFFF. El *slot* 1 no contiene nada, pero el 2 contiene una página de ROM, ocupando las posiciones desde &H4000 hasta &H7FFF, y una página de RAM, desde &H8000 hasta &HBFFF.

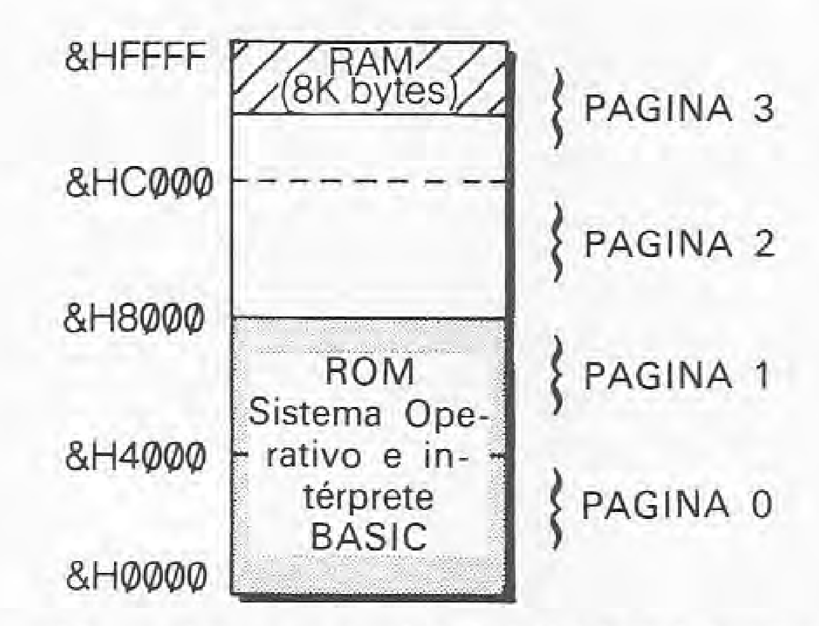


Figura 10.1.—*Mapa de memoria del* slot *0* (slot *del sistema*).

El ordenador completa entonces su mapa de memoria usando páginas de distintos *slots*, tratando la RAM de los *slots* 0 y 2 como un área continua, que tendría 32K bytes de larga. Si el ordenador necesita utilizar *software* de control de disco, simplemente busca sus instrucciones en la página 1 del *slot* 2, en lugar de en la página 1 del *slot* 0. ¿Cómo consigue acceso el sistema MSX a páginas de memoria que están en *slots* diferentes del *slot* del sistema? Bien, aquí es donde el registro de selección de *slot* representa un papel importante en el control del ordenador. Este registro, como probablemente recordará, es el A del PPI.

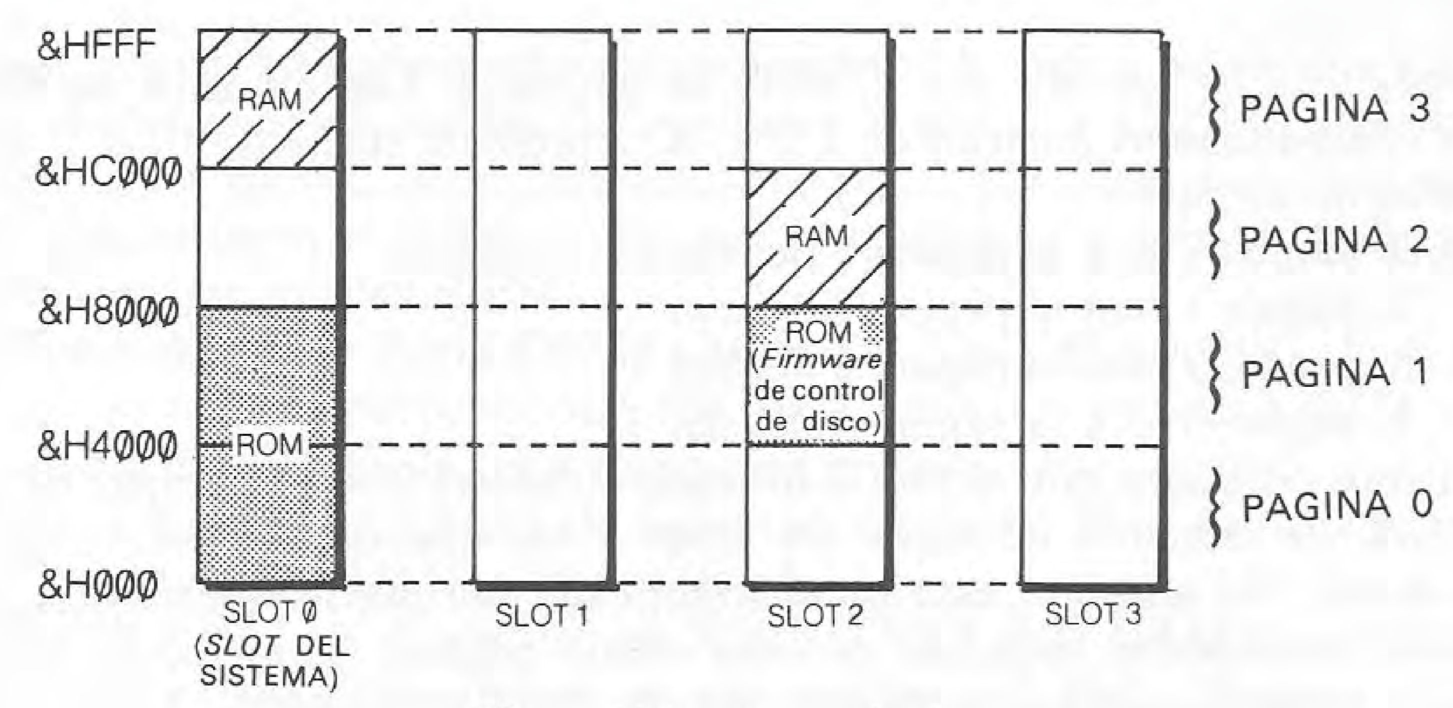


Figura 10.2.—*Banco de* slots.

Registro de selección de "slot"

Este registro informa al ordenador de qué *slot* ha de usarse para acceder a una cierta página de memoria. Los bits 0 y 1 contienen el *slot* de la página de memoria 0; los bits 2 y 3, el número de *slot* para la página 1; los 4 y 5, el número

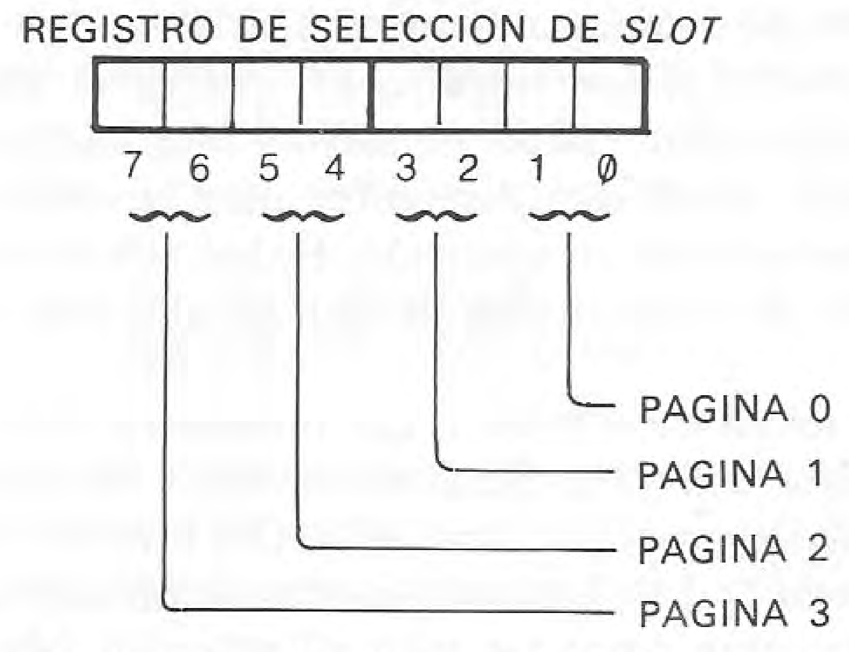


Figura 10.3.—*Registro de selección de* slot *(1): funciones de los bits.*

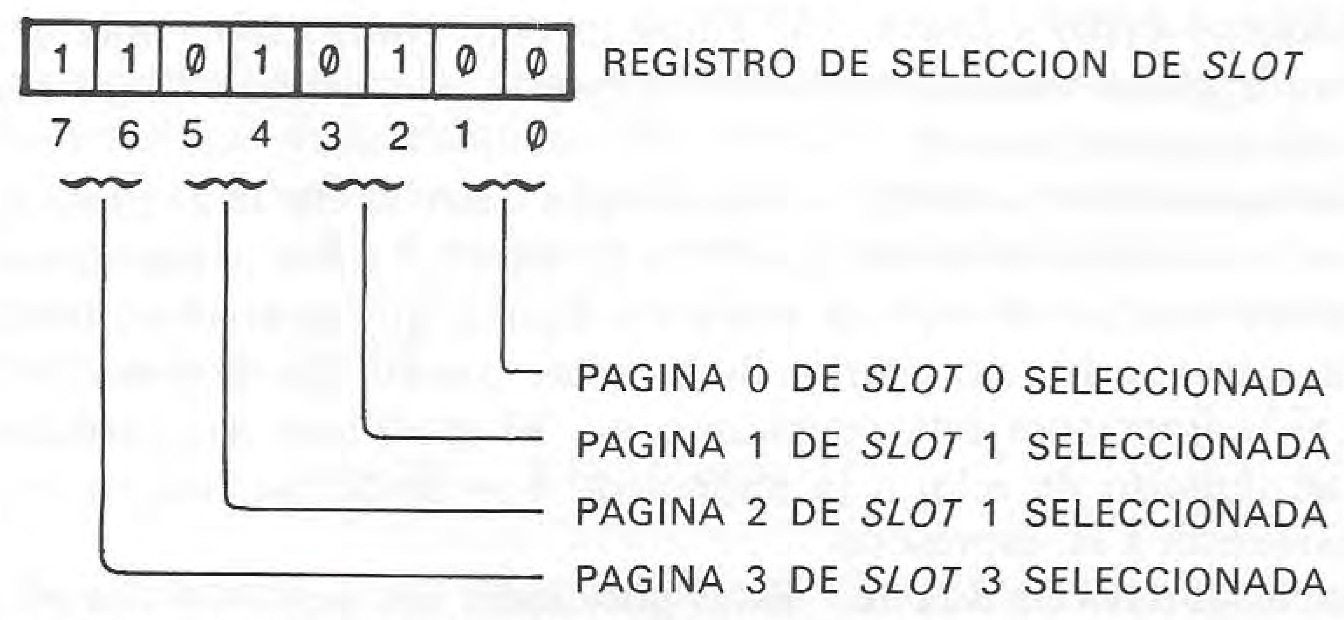


Figura 10.4.—*Registro de selección de* slot *(2): ejemplo de selección.*

para la página 2, y los bits 6 y 7, el de la página 3. Los números de *slot* se almacenan como números binarios de 2 bits; 00 representa el *slot* 0; 01, el 1, etc.

Veamos un ejemplo:

Aquí, la página 0 será la página 0 del *slot* 0;
la página 1 será la página 1 del *slot* 1;
la página 2 será la página 2 del *slot* 1;
la página 3 será la página 3 del *slot* 3.

Si tenemos ocupados por memoria los cuatro *slots*, el Z-80 será capaz de seleccionar sus 64K de memoria de entre los 256K a los que tiene acceso a través del *slot* del sistema. No obstante, esto no es todo; cada *slot* puede tener asociados otros cuatro más, conteniendo cada uno de ellos cuatro páginas de memoria. Estos *slots*, que surgen de los originales, se llaman *SLOTS* SECUNDARIOS. Los cuatro *slots* que pueden seleccionarse directamente a través del registro de selección de *slots* se llaman *SLOTS* PRIMARIOS. El control de la estructura de *slot* secundaria se escapa al alcance de este libro, y sólo se menciona para que el lector se convenza del potencial del sistema de *slot* para expansiones futuras.

La distribución exacta de la RAM es similar en todos los ordenadores MSX; el BASIC examinará la RAM del ordenador y buscará el área continua de RAM mayor, que comience en la posición &HFFFF. Por tanto, la RAM presente en cualquier *slot* de la máquina puede llegar a formar parte de la memoria usada por el BASIC. Parte de esta memoria, en las posiciones altas de RAM, se aporta para lo que se llama ESPACIO DE TRABAJO *(system workspace)*, que es el área de memoria RAM utilizada por el ordenador como "papel en sucio". Más adelante abundaremos sobre este tema. El resto de la RAM está disponible para el texto del programa BASIC y para las variables generadas en su ejecución. Habrá advertido que la VRAM nunca aparece formando parte de la estructura de *slot*, ya que sólo el VDP puede acceder a ella directamente.

Ahora veremos el mapa de IN/OUT del ordenador MSX; o sea, cómo están organizadas y distribuidas entre los diferentes circuitos integrados las 256 posiciones a las que podemos acceder con los comandos OUT e INP que estudiamos en el capítulo 9. Las posiciones de I/O listadas a continuación son las dadas en la especificación MSX; todas ellas eran correctas para mi máquina, pero pueden diferir si algunos fabricantes deciden cambiar el mapa[2].

Posiciones &H00 a &H80. No se utilizan en la especificación MSX actual.

Posiciones &H80 a &H88. Se emplean para controlar el interfaz RS232 que incorporan algunos ordenadores MSX. Como no lo utilizan todas las máquinas, no profundizaremos más en él.

Posiciones &H90 a &H91. Controlan el interfaz de la impresora, y por ello se usan con los comandos BASIC, LLIST y LPRINT. Tienen pocas aplicaciones, a menos que planee escribir rutinas en código máquina que controlen directamente la impresora a través de este puerto. Si lo hace, cuando lea la posición &H90 el bit 1 indica si la impresora está ocupada o no. Al modificar esta posición, el ordenador envía un impulso de reloj a la impresora. Los datos escritos en la posición &H91 se transmitirán a la impresora.

Posiciones &HA0 a &HA2. Estas posiciones nos permiten controlar directamente el PSG a través de los comandos INP y OUT. El verdadero valor de estas posi-

ciones surge cuando estamos escribiendo programas en código máquina, ya que será posible controlar el PSG con las instrucciones Z-80 IN y OUT. Sin embargo, es posible usarlas en BASIC, como enseguida veremos; por otra parte, los principios aquí señalados tendrán el mismo valor cuando vayamos a escribir programas en código máquina que utilicen el PSG.

La posición &HA0 se llama DIRECCION DE LATCH del PSG. Para escribir un valor en un registro particular del PSG utilizamos un comando OUT que nos permita colocar el número de registro en esta posición. Por ejemplo, para escribir un valor en el registro de control de amplitud del canal 1, el registro 8, primeramente escribiríamos el valor 8 en esta posición.

La posición &HA1 es el registro de ESCRITURA DE DATOS del PSG, y se usa cuando queremos escribir datos en un registro del PSG. Después de fijar el número de registro en la posición &HA0 escribimos en esta posición los datos que queremos poner en el regisgro. Por eso, para escribir el valor 15 en el registro 8 del PSG podríamos usar los comandos siguientes:

```
10 OUT &HA0,8
20 OUT &HA1,15
```

La posición &HA2 se llama registro de LECTURA DE DATOS del PSG, y se utiliza para leer datos de los registros del PSG. Es una capacidad que se echa de menos en los comandos de sonido BASIC. Para leer, por ejemplo, el contenido actual del registro 8 del PSG se emplean las siguientes líneas de texto (imprimirán el contenido del registro especificado en el *latch* de direcciones del PSG):

```
10 OUT &HA0,8
20 PRINT INP(&HA2)
```

Evidentemente ésta es una característica útil, que nos permite conocer en cualquier momento los contenidos de los registros del PSG. También nos permite acceder directamente a los dos registros de entrada/salida *(input/output)* del PSG, que normalmente no son accesibles.

Posiciones &H98 a &H99. Permiten acceder directamente al VDP sin utilizar el comando VDP. Tienen muchas aplicaciones para el programador en código máquina. También nos permiten acceder directamente a la VRAM sin usar VPOKE o VPEEK.

Otra vez, las posiciones exactas de estas dos direcciones pueden estar colocadas en varios sitios dependientes de la máquina. Sin embargo, en este ejemplo las posiciones del mapa IO, que corresponden a las posiciones de lectura y escritura del VDP, se almacenan en las posiciones 6 y 7, respectivamente, de la ROM MSX.

Para leer el registro de estado del VDP llevamos a cabo una instrucción *input* de la posición &H99. La línea BASIC

```
PRINT INP(&H99)
```

leerá el registro de estado del VDP. Para saber qué efectos produce la lectura de este registro en su contenido consulte el capítulo 6. Es muy importante leer siempre

este registro antes de intentar modificar los registros del VDP. Este proceso informa al VDP de que se prepare para recibir información, así como para la operación de escritura.

Escribir un registro del VDP no es complicado. Hay tres operaciones implicadas, y se exponen en la rutina escrita a continuación, que proporciona el método general de modificar un registro del VDP:

```
100 SIN=INP(&H99)
105 REM operacion de lectura
110 OUT &H99,VALOR
115 REM byte a escribir
120 OUT &H99,(NUMERO DE REGISTRO+128)
```

Valor es el byte de datos que se quiere escribir en el registro del VDP, y por eso debe estar entre 0 y 255. La segunda operación de escritura envía el número de registro modificado al VDP. Como ejemplo concreto, las líneas siguientes escriben el valor 18 en el registro 7 del VDP:

```
100 sin=INP(&H99)
110 OUT &H99,18
120 OUT &H99,135
```

¿Cómo se accede a la RAM de video a través de estas direcciones? Es algo más complicado que acceder a los registros del VDP, aunque bastante fácil.

Empecemos escribiendo un valor en la VRAM. También aquí es una buena costumbre leer la posición &H99 antes de iniciar una operación de escritura en VRAM. Para indicar al VDP a qué posición de VRAM queremos enviar datos tenemos que modificar dos veces la posición &H99. Entonces enviamos el byte de datos que pretendemos colocar en esa posición a la dirección de IO &H98. El efecto que produce escribir un valor particular en una cierta dirección de VRAM en cada modo se expuso en el capítulo 6. Las líneas siguientes apuntan la técnica básica para modificar un byte de VRAM:

```
100 SIN=INP(&H99)
110 OUT &H99,DIRECCION DE VRAM MOD 256
120 OUT &H99,(direccion de VRAM¥256)+64
130 OUT &H98,VALOR
```

Observe la necesaria manipulación de la dirección de VRAM. Recuerde que el signo significa división entera. Como ejemplo real, la rutina listada a continuación escribirá el valor de 65 en la dirección 0 de VRAM. El mejor modo para demostrarlo en BASIC es el modo 0, ya que la dirección 0 de VRAM en este modo corresponde a la primera posición de la tabla de nombres, y por eso veremos el efecto en la pantalla:

```
100 sin=INP(&H99)
110 OUT &H99,0
120 OUT &H99,64
130 OUT &H98,65
```

Raramente queremos modificar una sola posición de VRAM; el VDP lo tiene en cuenta y, cada vez que un byte ha sido colocado en una posición de VRAM, el siguiente byte enviado a la dirección &H98 será escrito en la posición posterior de VRAM. Por tanto, en el ejemplo que acabamos de ver, el comando OUT &H98 escribiría el valor en la posición 1 de VRAM, suponiendo que entre tanto no se haya emitido ningún comando de acceso al VDP. Esta característica, que se llama AUTO INCREMENTO, posibilita al programador en código máquina escribir secuencias de bytes en la VRAM sin tener que repetir cada vez los valores a la dirección.

Evidentemente, esta característica sólo tiene utilidad si los bytes a escribir en la VRAM tienen posiciones contiguas en memoria.

Para leer un valor de la VRAM se emplea un método similar. Simplemente sustituimos la operación OUT, que en la línea 130 escribe la dirección &H98 por una instrucción INP que lea esa posición. Así, la rutina siguiente leerá el valor actual contenido en la posición 0 de la VRAM:

```
100 sin=INP(&H99)
110 OUT &H99,0
120 OUT &H99,0
130 PRINT INP(&H98)
```

Posiciones &HA8 a &HAB. Estas posiciones son las de los registros del PPI, vistas en el capítulo 9.

Posiciones &HB0 a &HB3. Se utilizan en algunas máquinas para controlar memoria adicional.

Posiciones &HB8 a &HBB. En algunos ordenadores controlan la operación del lápiz óptico.

Lo expuesto completa este estudio de las áreas del mapa IO de utilidad para el programador MSX (algunos fabricantes pueden ignorar estas directrices). No obstante, las posibilidades que de estas direcciones ofrece al programador en código máquina de los distintos componentes del sistema son bastante útiles y no deben ser ignoradas por nadie que pretenda programar los ordenadores MSX.

Organización de la RAM

La mayoría de la RAM del ordenador es utilizada por el BASIC para almacenar programas y variables. En el capítulo 4 vimos someramente cómo están escritos en RAM los programas. Ahora lo veremos de nuevo, con más detalle. Empezaremos averiguando dónde se almacena en memoria el programa BASIC. La manera más fácil de hacerlo es evaluar la expresión:

```
PRINT 65536-(n*1024)
```

donde *n* es el número de K de memoria que posee la máquina; por ejemplo, para un ordenador MSX de 16K tendremos un valor de 49152.

Veamos ahora un ejemplo de cómo se almacena el típico programa corto en memoria.

El programa es:

```
10 REM test
20 PRINT "Hola"
30 END
```

Si utilizamos un bucle FOR ... NEXT para conocer los valores contenidos en RAM desde la posición 49152 en adelante, obtendríamos:

49152	0	
49153	12	comienzo de la línea 10
49154	192	es un puntero de dos bytes que señala el principio de la línea 20
49155	10	byte menos significativo del número de línea
49156	0	byte más significativo del número de línea
49157	143	índice de la palabra REM
49158-49162		texto de la sentencia REM
49163	0	fin de la línea 10
49164	26	comienzo de la línea 20 igual
49J65	192	que para la línea 10
49166-49177		resto de la línea 20
49178	32	comienzo de la línea 30 igual
49179	192	que para la línea 10
49184	0	dirección señalada por el puntero
49185	0	del principio de la línea 30

Los dos números de las direcciones 49153 y 49154 forman un solo número de 16 bits. La dirección que indican puede calcularse por

```
PRINT 12+256*192
```

y vemos que señala el comienzo de la línea 20. Los siguientes 2 bytes son el número de línea, organizándose otra vez como un número de 16 bits, siendo el primer byte el menos significativo. Este número de línea puede calcularse por un proceso similar al anterior. A continuación sigue el texto de la línea, terminado con un cero que marca el final. Este modelo general se repite para cada línea del programa

hasta llegar a la última. El par de ceros en las posiciones 49184 y 49185 indican al ordenador que es la última línea de texto escrita en BASIC.

Si alteramos las posiciones 49153 y 49154 colocando en ellas un valor 0, el ordenador "pierde" el programa; sin embargo, no tiene el mismo efecto que el comando NEW: éste borra toda la memoria de tal manera que FRE(0) tiene el mismo valor que muestra cuando se conecta el ordenador. Después de modificar estas dos posiciones, FRE(0) nos daría un valor que indicaría que el programa está aún presente. Si ponemos otro 0 en la posición 49152, el ordenador es incapaz de ejecutar el programa, incluso aunque todavía pueda listarlo mediante un comando LIST.

Almacenamiento de variables

Después de escribir un programa BASIC, éste puede ejecutarse y generar sus variables, que se almacenan en la RAM tras el programa o, en el caso de distintas variables de cadena, en el "espacio para cadenas" asignado para este cometido por el ordenador. Las observaciones siguientes se han hecho estudiando el área de RAM que sigue al programa, y por ello acepto la total responsabilidad si comete algún error de interpretación.

Variables de cadena

Cuando se asigna la variable explícitamente, por ejemplo:

```
a$="abcdefg"
```

el área de memoria —a la que apunta el comando VARPTR— que contiene a la variable se estructura de la manera siguiente:

3	indica el tipo de variable
n$	primera letra del nombre de variable
*n*1$	segunda letra del nombre de variable
L	longitud de la cadena
bajo	dirección del último punto del texto del programa
alto	donde se hizo la asignación de la variable. Apunta a la primera letra de la cadena constante asignada a la variable

La última dirección puede evaluarse de modo similar al puntero de direcciones en el texto de programa BASIC. Como ejemplo de su uso, en el anterior señalaría la posición en el programa de la letra "a" al principio de la cadena "abcdefg".

Si la asignación de la variable dentro del programa es de la forma

```
a$=b$
```

aquí el puntero señalará la primera letra de la cadena asignada a b$, si a b$ se le asignó una cadena constante en el programa. Interesa señalar que cuando tecleamos algo como

```
a$="MSX"
```

en una línea de programa, se inserta un byte extra en la línea, indicando que se ha realizado una asignación. Asimismo los bytes del puntero señalan ahora el área de memoria en el espacio para cadenas donde están almacenadas las letras que componen la cadena. Por este motivo, en el ejemplo anterior el puntero indicaría la posición de memoria en el espacio para cadenas que contiene la letra "A". El valor retornado por VARPTR(a$) en dicho ejemplo es la dirección de la longitud de la cadena en la tabla de variables.

Matrices de cadena

El bloque de memoria que contiene los datos acerca de los contenidos de la matriz de cadena se muestra a continuación. No es seguro el significado de los bytes cuarto y quinto de la tabla, pero podrían indicar al ordenador que ese bloque de memoria representa una matriz de cadena. Un punto a destacar aquí, que también es aplicable a las variables de cadena normales, es que si se hace una asignación desde el modo directo el puntero de la variable concierne a varios puntos dentro de la cadena:

3	identificador
n$	primer carácter del nombre
*n*1$	segundo carácter del nombre
?	significado desconocido
?	significado desconocido
dim	número de dimensiones
eme	byte menos significativo para el número total de elementos
ema	byte más significativo para el número de elementos
lon0	longitud del primer elemento de la matriz
dir0	byte menos significativo de la dirección del contenido del primer elemento
dir0ma	byte más significativo de la dirección del contenido del primer elemento
	(Los tres últimos datos se repiten para cada elemento de la matriz.)

Cuando decimos "primer elemento de la matriz" queremos decir el elemento 0. Por eso el primer elemento de la matriz a$ es a$(0). Si los contenidos de la longitud y de la dirección de un elemento de la matriz son 0, indica que el elemento en cuestión no tiene un valor asignado.

Variables enteras

Son las variables numéricas más simples, y pueden ser útiles cuando deseemos dar y recibir valores de las rutinas en código máquina. Se implantan en memoria como se muestra a continuación:

2	identificador
n$	primer carácter del nombre
*n*1$	segundo carácter del nombre
valmen	byte menos significativo del valor numérico
valmas	byte más significativo del valor numérico

El valor que contiene una variable entera puede calcularse fácilmente mediante el siguiente método:

```
PRINT PEEK(By. menos signif.)+256*PEEK(By. mas signif.)
```

Las otras variables numéricas no son tan fáciles de comprender y utilizar, y por eso no entraremos en detalles.

Variables reales

Hay dos tipos de variable real: las de "doble precisión" y las de "simple precisión". La manera en que el ordenador almacena estos dos tipos de variables es muy similar:

8	identificador
n$	primer carácter del nombre
*n*1$	segundo carácter del nombre
exp	exponente
mantisa	una mantisa de 4 bytes para las variables de "simple precisión", estando colocado en el último lugar de la tabla el byte menos significativo (Una mantisa de 7 bytes para las variables de "doble precisión", siendo otra vez el último byte de la tabla el el menos significativo.)

El exponente del número se almacena en forma codificada, como el exponente +65. Así, un exponente de 1 se codifica como 66. Si el exponente es negativo, se efectúa otra adición más, siendo representados los exponentes negativos añadiendo 128 al valor que tendría un exponente positivo de la misma magnitud.

Matrices enteras

Tienen una estructura similar a los de cadena, como puede apreciarse en la tabla:

2	identificador
n$	primer carácter del nombre

*n*1$	segundo carácter del nombre
?	significado desconocido
?	significado desconocido
dims	número de dimensiones
eme	byte menos significativo del número de elementos
ema	byte más significativo del número de elementos
ele0men	byte menos significativo del elemento 0
ele0mas	byte más significativo del elemento 0

(Los dos últimos bytes se repiten para el resto de los elementos de la matriz.)

Matrices reales

Se estructuran de un modo similar a las matrices enteras, pero con un identificador de 8 y números de doble o simple precisión como elementos de la matriz. La estructura de estos elementos es la misma que para los números reales ordinarios.

Espacio de trabajo del sistema

El ordenador MSX es un sistema muy complicado, y necesita utilizar algo de RAM para necesidades interiores del sistema. La RAM usada por la ROM MSX se llama "espacio de trabajo del sistema", y su máquina nunca tiene tanta memoria para sus programas BASIC como usted podría pensar. Cuando teclea sus programas, el ordenador se asegura, antes de intentar ejecutarlos, de que tiene suficiente espacio para realizar sus propias tareas.

En esta sección se menciona someramente un par de áreas de memoria que pueden serle útiles cuando programe su ordenador MSX. Evidentemente no es una visión exhaustiva del sistema, pero se exponen unos cuantos apuntes que justifican la necesidad del espacio de trabajo del sistema. Si desea conocer esta área, puede escribir un programa simple que le permita visualizar, a la vez, bloques de 20 bytes en la pantalla, de tal modo que pueda ver si alguno de éstos cambia de valor con el tiempo (por ejemplo, el programa siguiente es un medio simple de examinar cualquier bloque de memoria de esta manera). Después de haberse visualizado los primeros 20 bytes, pulse cualquier tecla para verlos de nuevo: evidenciará cualquier cambio en el valor de un byte. Para dar un nuevo valor de comienzo vuelva a ejecutar el programa:

```
10 INPUT COMIENZO
20 FOR I=COMIENZO TO COMIENZO+20
30 PRINT I;" ";:J=PEEK(I)
40 PRINT J;
```

```
50 IF J>31 AND J<127 THEN PRINT CHR$(J)
ELSE PRINT
55 NEXT I
60 I$=INPUT$(1):CLS:GOTO 20
```

El espacio de trabajo del sistema está compuesto de todas las direcciones de RAM por encima de la 62336; por tanto, tenga cuidado cuando desee alterar esta región de la memoria.

Definiciones de las teclas de función

Las cadenas asignadas a las teclas de función se almacenan en el espacio de trabajo del sistema desde la dirección 63615 hasta la 63774. Por ello es posible alterar las cadenas que contienen las teclas accediendo directamente a la memoria. También es posible almacenar en cinta las definimiones de teclas utilizando BSAVE, una vez conocidas las direcciones.

"Buffer" de entrada

El área de memoria desde la posición 64496 a la 64576 la utiliza el ordenador para guardar un registro de las teclas pulsadas. Esta área de memoria se llama "*buffer* de entrada".

Comienzo de RAM

El comienzo de la RAM que usa el BASIC está contenido en las posiciones 64584 y 65585. En consecuencia, el comienzo de la RAM disponible puede calcularse mediante

```
PRINT PEEK(64584)+256*PEEK(64585)
```

Comienzo del espacio de trabajo del sistema

Esta dirección está en las posiciones 64586 y 64587. El principio del espacio de trabajo puede calcularse de modo similar al de comienzo de la RAM.

Contador de tiempo (TIME)

Las posiciones 64670 y 64671 contienen la variable TIME, que se incrementa regularmente. Por tanto, colocar a 0 estas dos posiciones tendrá el mismo efecto que poner a 0 la variable TIME con una sentencia como

```
TIME=0
```

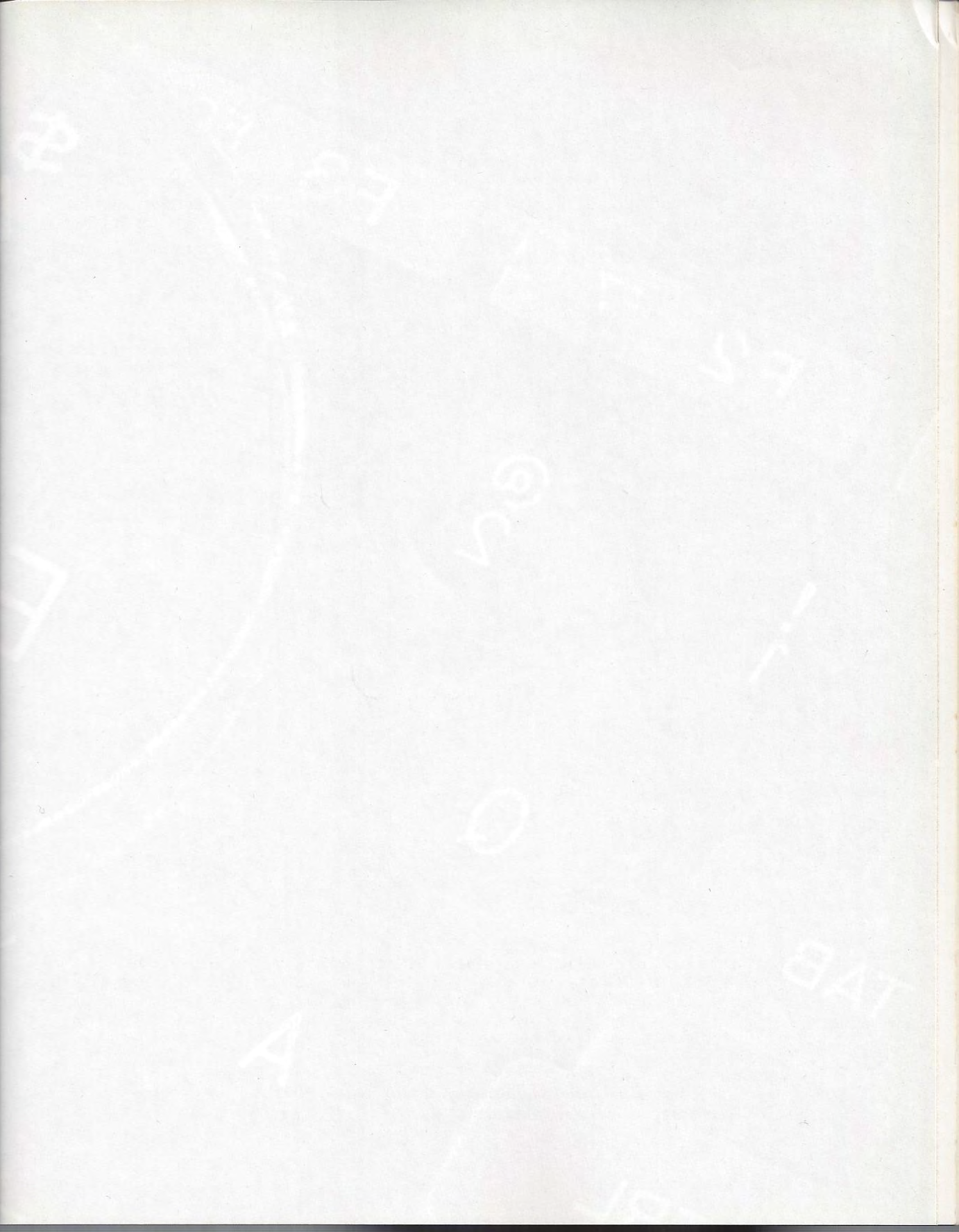
Obviamente esto es muy útil, ya que nos permite modificar la variable TIME desde dentro de un programa en código máquina.

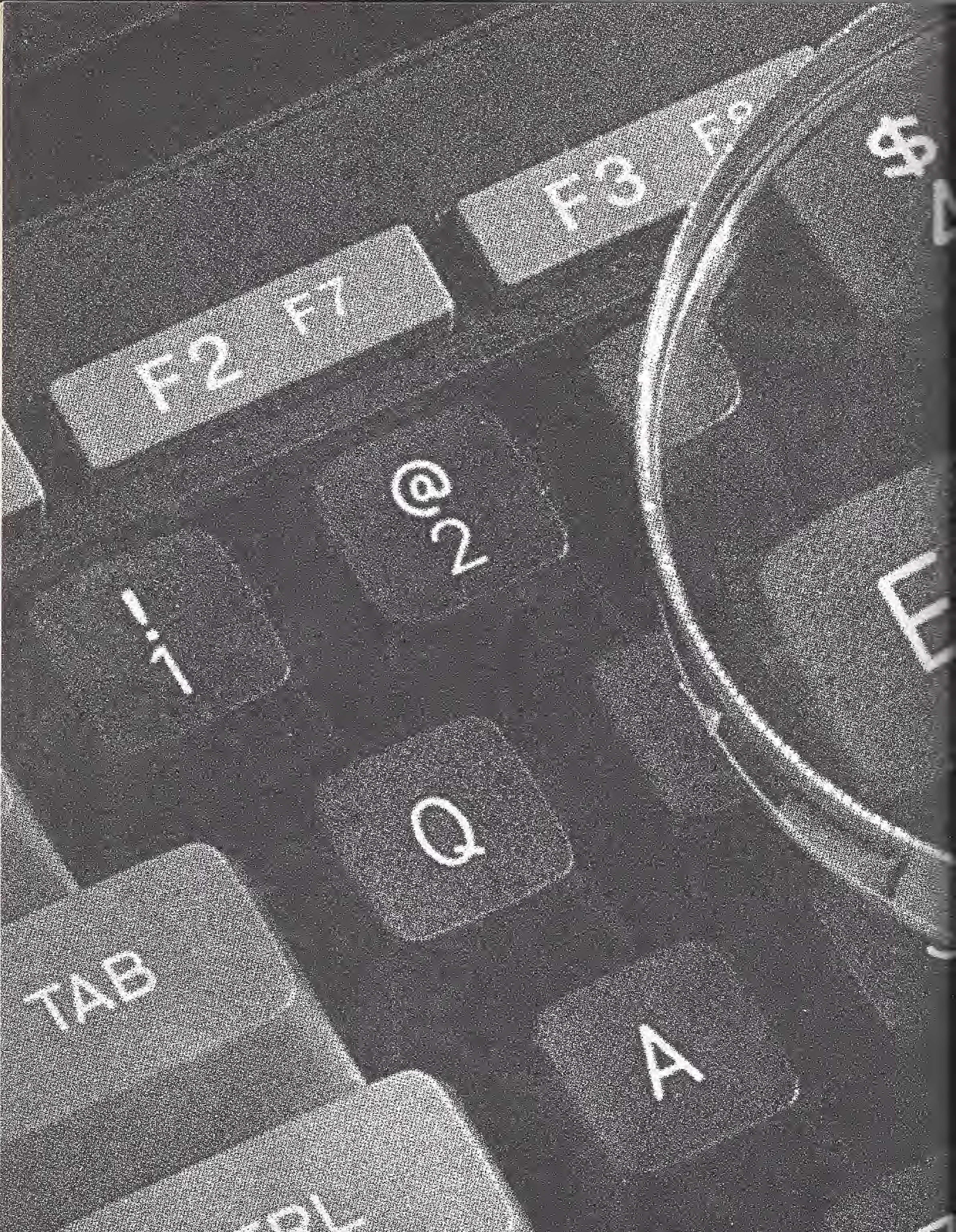
Estas son las áreas del espacio de trabajo del sistema más útiles para el programador BASIC. En el próximo capítulo examinaremos un par de rutinas simples que pueden tener interés, y más adelante discutiremos algo acerca del método de programación BASIC.

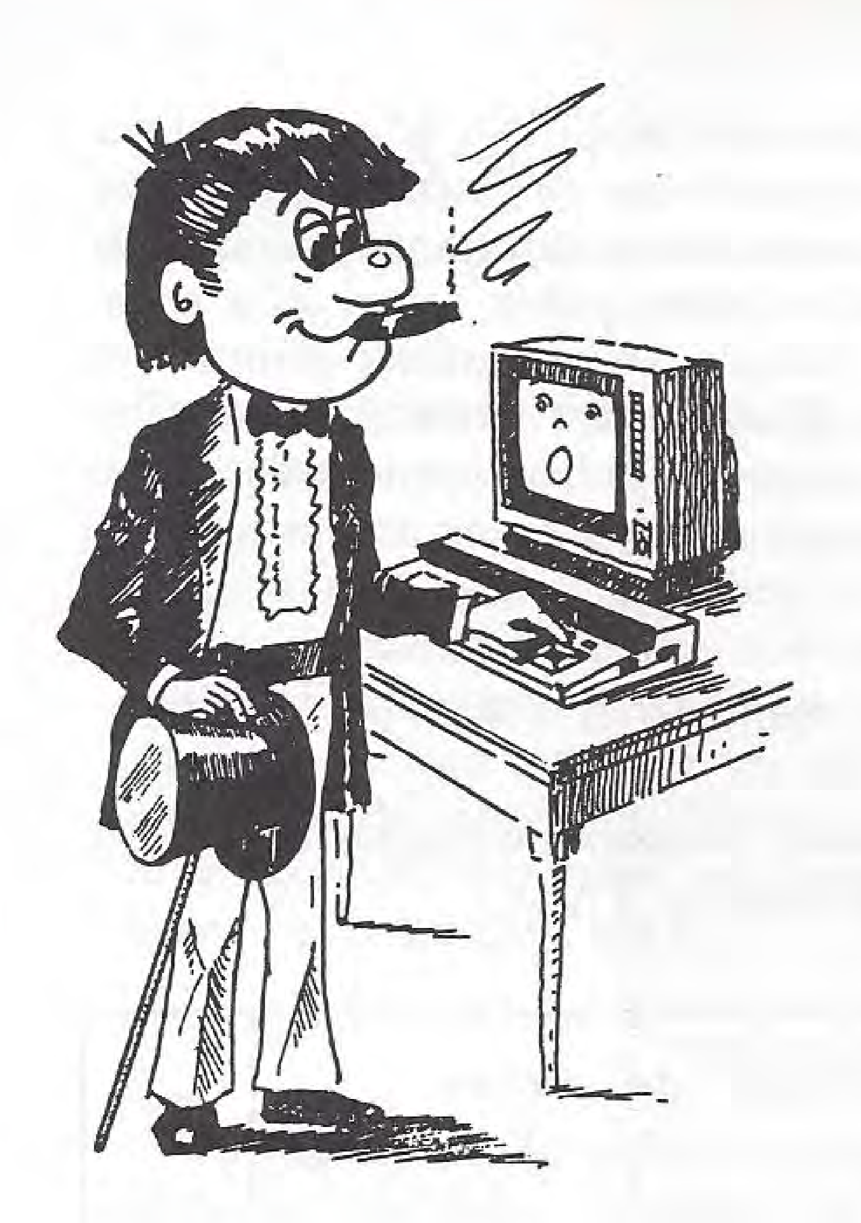
NOTAS DEL CAPITULO

1.—Aunque la CPU Z-80 sólo puede manejar 64K de memoria, de los cuales 32 los ocupa el sistema operativo, utilizando los *slots* puede obtenerse más memoria, llegando hasta 1 *Mbyte* utilizando también los *slots* secundarios. Sin embargo, no es posible manejar los 20 *slots* a la vez, sino que solamente se puede acceder a cuatro páginas suyas a un tiempo. Es posible utilizar toda esta memoria cambiando de un *slot* a otro mediante rutinas en código máquina.

2.—En el estándar MSX se aconseja utilizar subrutinas de la ROM para realizar tareas como las que se exponen en este capítulo, subsanando de esta manera las posibles diferencias de *hardware* entre los distintos ordenadores.




F2 F7
F3
@
2
!
1
Q
TAB
A
E



11 Estilo de BASIC y rutinas simples

El objetivo de este capítulo es doble; primeramente, introducir algunas rutinas simples pero útiles para sus programas, y en segundo término, desarrollar las técnicas implicadas en la creación de programas BASIC de manera que esté tan libre de errores y sea tan legible como podamos. Puede que esto no le parezca excesivamente importante en este momento, pero puede que lo sea si se presenta el caso de que tenga que volver a un programa escrito algunos meses antes porque han surgido problemas de desarrollo; en tal caso apreciará la legibilidad de sus programas.

Sin embargo, empezamos viendo un par de rutinas simples de utilidad. La primera de ellas soluciona un problema que se da con frecuencia en las máquinas MSX; retornan al modo de texto siempre que se ejecuta una instrucción INPUT estando en modo gráfico. ¡Esto será extremadamente molesto si tenemos gráficos en la pantalla! La rutina se basa en las técnicas que vimos primero en el capítulo 6.

Rutina de entrada de datos para uso general

Esta subrutina necesita que se declaren algunas variables para su uso. Son las siguientes:

CFO	color de fondo
CTE	color del texto
X,Y	posición en la pantalla gráfica donde quiera que se introduzcan los datos
L	longitud de la cadena a introducir.

El último parámetro es la longitud de la cadena más larga que la rutina deberá aceptar como entrada. Esta versión solamente aceptará letras mayúsculas pulsadas en el teclado, pero puede modificarse para que acepte otros caracteres alterando la línea 1030 de la subrutina. Por otra parte, usted no puede pulsar *return* para abandonar la subrutina, sino que tiene que haber tecleado algún carácter primero. Si comete un error mientras introduce los datos, utilice BS para borrar los caracteres equivocados. Cuando abandone la rutina, A$ contiene la cadena introducida. Como ejemplo de una modificación, podría alterar la subrutina para que acepte números alterando la línea 1030;

```
1030 IF (G<47 OR G>57) AND G<>13 AND G<>8 THEN GOTO 1020
```

La cadena estará ahora compuesta de números conteniendo dígitos del 0 al 9. Para convertirla en un valor numérico, utilice el comando VAL.

```
1000 REM Subrutina de entrada de datos
1010 OPEN "GRP:" AS #1
1015 A$=""
1020 G$=INPUT$(1):FL=0
1025 G=ASC(G$)
1030 IF (G<65 OR G>96)AND G<>32 AND G<>13
AND G<>8 THEN GOTO 1020
1035 IF G=8 AND A$="" THEN GOTO 1015
1045 IF G=8 THEN FL=1
1050 IF G=13 AND A$="" THEN GOTO 1020
1060 IF G=13 THEN CLOSE#1:RETURN
1070 IF FL=0 THEN A$=A$+G$
1080 IF FL=1 THEN
C$=MID$(A$,1,LEN(A$)-1):PRESET(X,Y)
1085 COLOR CFO:PRINT#1,A$:COLOR CTE
1086 PRESET(X,Y):PRINT#1,C$;:A$=C$
1090 A$=LEFT$(A$,L):PRESET(X,Y)
PRINT#1,A$:GOTO 1020
```

Esta rutina puede, desde luego, modificarse para usarla en modos de texto, donde todavía ofrece más ventajas sobre las sentencias INPUT normales, ya que permite a los usuarios teclear números cuando se esperan cadenas y viceversa. El programa siguiente le provee de una demostración de la rutina anterior. Si quiere colocar la rutina en cualquier otro lugar de la memoria, altere los números de línea en la sentencia GOSUB.

```
10 CFO=4:CTE=15:L=5:X=100:Y=100
20 SCREEN 3
30 GOSUB 1000
```

```
40 PRINT A$
50 END
```

Comprobación de teclas específicas

La rutina siguiente ofrece al programador un método de uso general para comprobar que el usuario ha pulsado una tecla específica. Las teclas RETURN y SPACE se usan frecuentemente para controlar el desarrollo de un programa; esta rutina permitirá al programador tener una rutina que compruebe todas las teclas correctas. Al entrar en esta rutina, la variable TEST$ contendrá todas las letras permitidas en ese punto del programa. Pueden incluirse en ella letras minúsculas y mayúsculas. Si se necesita comprobar caracteres como RETURN, se pueden insertar en TEST$ usando el comando CHR$( ).

```
TEST$="AaBbCc"+CHR$(13)
```

Al regresar de la rutina, la variable PS contiene la posición, dentro de TEST$, de la tecla pulsada. La rutina no finaliza hasta que se ha pulsado una tecla permitida:

```
1000 REM rutina de comprobacion
1005 REM de teclas
1010 G$=INPUT$(1)
1020 IF INSTR(TEST$,G$)=0 THEN
GOTO 1010
1030 PS=INSTR(TEST$,G$)
1040 RETURN
```

Como demostración de su uso, pruebe el siguiente programa (que comprobará las letras S o N, en mayúsculas o minúsculas):

```
10 TEST$="SsNn"
20 GOSUB 1000
30 IF PS>2 THEN PRINT "No"
ELSE PRINT "Si"
40 GOTO 20
```

Salvar imágenes de pantallas

Las siguientes rutinas le permitirán salvar en cinta distintas secciones de la RAM de video, y volverlas a cargar en la VRAM cuando quiera. Obviamente, si archiva

las áreas apropiadas, es posible salvar la imagen actual de la pantalla. Esto tiene una utilidad particular si la imagen de la pantalla tarda mucho tiempo en dibujarse.

Aquí se describen dos métodos: uno, usando PRINT#, funcionará con todos los micros MSX y salvará completamente la pantalla en cualquier modo. Sin embargo, es bastante lento. El segundo, que usa BSAVE, necesita un área de memoria del mismo tamaño que el área de VRAM mayor que haya que archivar; esta área actúa como *buffer* de salida de datos. Aunque es más rápido, este método puede causar problemas en los modos gráficos de los ordenadores MSX con sólo 16K de memoria. No obstante, si no quiere salvar toda la VRAM sino sólo parte de ella, puede usar cualquiera de los dos métodos.

Como ejemplo, salvaremos la pantalla en modo 0 en la cinta, usando cada uno de estos dos métodos. De todas maneras, en cualquiera de ellos puede aumentar la velocidad del proceso si utiliza la velocidad de transferencia de 2.400 baudios para la operación de escritura. Vea el capítulo 4 si desea más detalles.

La primera rutina, usando PRINT#, tarda 1 minuto y 20 segundos en archivar en la cinta la tabla de nombres y la tabla de patrones del modo 0:

```
1000 REM archivado de pantalla
1010 NM%=BASE(0)
1020 PM%=BASE(2)
1030 OPEN "CAS:SCREEN" FOR OUTPUT AS #1
1040 FOR I%=NM% TO NM%+959
1050 C%=VPEEK(I%):PRINT#1,CHR$(C%);
1060 NEXT
1070 FOR I%=PM% TO PM%+2047
1080 C%=VPEEK(I%):PRINT#1,CHR$(C%);
1090 NEXT:CLOSE#1
1100 RETURN
```

Esta rutina salvará las áreas de VRAM que sirven para definir la pantalla en modo 0. Aunque esto es útil, es una buena idea llamarla desde el programa mientras está en uso el modo 0, y será almacenada la pantalla visualizada en ese momento.

La siguiente rutina realiza la tarea de volver a cargar la pantalla.

```
2000 REM carga de pantalla para el
modo 0
2010 NM%=BASE(0):PM%=BASE(2)
2020 OPEN "CAS:SCREEN" FOR INPUT AS #1
2030 FOR I%=NM% TO NM%+959
2040 A$=INPUT$(1,#1):VPOKE I%,ASC(A$)
2050 NEXT
2060 FOR I%=PM% TO PM%+2047
2070 A$=INPUT$(1,#1):VPOKE I%,ASC(A$)
2080 NEXT
2090 CLOSE #1:RETURN
```

Cuando quiera llamar a esta rutina, coloque primero el ordenador en modo de pantalla 0.

El segundo método, como se ha explicado anteriormente, transfiere el contenido de la VRAM a un bloque de RAM normal, y entonces usa el comando BSAVE para archivarla en la cinta. Este método es más rápido, en lo que se refiere a la escritura en la cinta, pero requiere más memoria, especialmente en modo 2.

El programa siguiente demuestra los principios que se utilizan. No hay nada que le impida utilizarlo como subrutina; pero, si lo hace, no olvide colocar el comando CLEAR de la línea 10 al principio del programa.

```
10 CLEAR 200,54999
20 J=55000
30 FOR I%=BASE(0) TO BASE(0)+959
40 POKE J,VPEEK(I%)
50 J=J+1:NEXT
60 J=56000
70 FOR I%=BASE(2) TO BASE(2)+2047
80 POKE J,VPEEK(I%)
90 J=J+1:NEXT
100 BSAVE "CAS:SCREEN",55000,59000
110 END
```

Para cargar esta pantalla de nuevo utilizamos BLOAD en lugar de INPUT$. Como siempre, usamos la rutina en modo 0.

```
10 CLEAR 200,54999!
20 SCREEN 0:BLOAD "CAS:SCREEN"
30 J=55000!:FOR I%=BASE(0) TO BASE(0)
+959:VPOKE I%,PEEK(J)
40 NEXT
50 J=56000!:FOR I%=BASE(2) TO BASE(2)
+2047
60 VPOKE I%,PEEK(J)
70 NEXT
80 END
```

Si usa estas rutinas para transferir otras áreas de VRAM, por ejemplo la tabla de patrones del modo 2, asegúrese de que tiene un *buffer* lo suficientemente grande en la RAM normal; si no lo hace, puede alterar accidentalmente parte del espacio de trabajo del sistema. De esta manera puede transferirse cualquier parte de la VRAM. Un uso singular es archivar los caracteres del modo 0 o del modo 1 salvando la

tabla de patrones apropiada. O podríamos salvar en cinta la tabla de patrones de *sprites*, archivando así las definiciones de *sprites* del programa.

Estilo de programación

Cualquiera puede escribir programas cortos con muy poca planificación. No obstante, tan pronto como empiece a escribir programas BASIC largos puede encontrar problemas debido a la falta de planificación del programa. En esta sección del capítulo se pretende desarrollar algunas técnicas que puedan ayudarle a escribir programas libres de errores, eficientes y fáciles de alterar y corregir.

Empecemos con la planificación del programa. Antes de conectar el ordenador siéntese y analice el problema dividiéndolo en sus partes constituyentes. Por ejemplo, supongamos que estamos escribiendo un programa para hacer un dibujo de una casa. Este problema puede fragmentarse en cuatro partes más pequeñas:

i Dibujar las paredes.
ii Dibujar el tejado.
iii Dibujar las ventanas.
iv Dibujar la puerta.

Este análisis en el que dividimos un solo, ligeramente largo, programa en otros cuatro más pequeños, en este caso, se conoce como de ARRIBA ABAJO *(top down)*. Cada uno de estos pequeños problemas puede desarrollarse independientemente, y podemos escribir una subrutina para hacer cada trabajo. Si las instrucciones BASIC necesarias para realizar este trabajo parecen demasiado largas, podemos dividir la tarea de nuevo en porciones más pequeñas, y escribir otra vez subrutinas para realizar estos pequeños trabajos.

Este método particular de diseño de programas tiene varias ventajas sobre el desarrollado simplemente delante del teclado. La primera es que cada subrutina desarrollada para resolver una parte determinada del problema especificado puede comprobarse independientemente, como hemos visto en el programa de demostración de la rutina de entrada de datos. La subrutina puede depurarse antes de colocarla definitivamente en el programa, y cualquier problema que surja después estará, por tanto, causado por algo que está fuera de esa sección del programa.

Un segundo punto a recordar es que si los programas se desarrollan como subrutinas que realizan tareas particulares dentro del programa, si no le gusta cómo se comporta esa parte del programa puede alterar sencillamente la subrutina apropiada.

En tercer lugar, después de que haya escrito unos cuantos programas, empezará a acumular un conjunto de subrutinas para realizar ciertas tareas comunes, tales como una entrada de datos, o rutinas de comprogación del teclado, como las que están listadas al comienzo del capítulo.

Cuando escriba sus subrutinas, empiécelas con una sentencia REM que le informe de lo que hace la subrutina, posiblemente listando también las variables que necesita ésta para trabajar correctamente. También es útil advertir en estas sentencias REM qué variables retornan los resultados cuando el control del programa regresa al cuerpo principal del mismo.

Nombres de variable

Ya hemos apuntado las "reglas" de la denominación de variables en el capítulo 3. Sin embargo, es bastante importante usar nombres de variables significativas siempre que sea posible, ya que podemos necesitar modificar el programa en el futuro. Por ejemplo, si usamos una variable de cadena con el nombre de un fichero a escribir en cinta, podríamos llamar a la variable a$. Pero si la variable se llamara nom$ o fich$, tendríamos mucha más información.

Por otra parte, esto presenta algunos problemas; recuerde que solamente las dos primeras letras de un nombre de variable son relevantes. Si no lo tiene en cuenta puede encontrarse con algunos programas muy confusos.

Las sentencias GOTO

El comando GOTO es muy útil, pero abusar de él puede conducir a programas teriblemente complicados. Si está intentando leer un programa de ordenador que tiene instrucciones GOTO cada cinco o seis líneas causando que el control del programa salte de una línea a otra, será muy difícil de entender.

Intente siempre restringir el uso de instrucciones GOTO en sus programas. Son necesarias; no hay manera en el BASIC MSX de eliminar totalmente el uso de estas instrucciones. Pero puede hacer programas que usen las instrucciones GOTO de una manera más legible. Es posible mantener el destino de una línea de un comando GOTO dentro de las seis o siete líneas siguientes; esto es bastante fácil, si ya hemos dividido el programa en subrutinas utilizando el método de programación TOP DOWN.

Otras cosas que ayudan a hacer más legible un programa son colocar la variable de control de un bucle detrás del comando NEXT, especialmente si el FOR está a varias líneas, o si se usan bucles FOR ... NEXT anidados.

Areas del programa

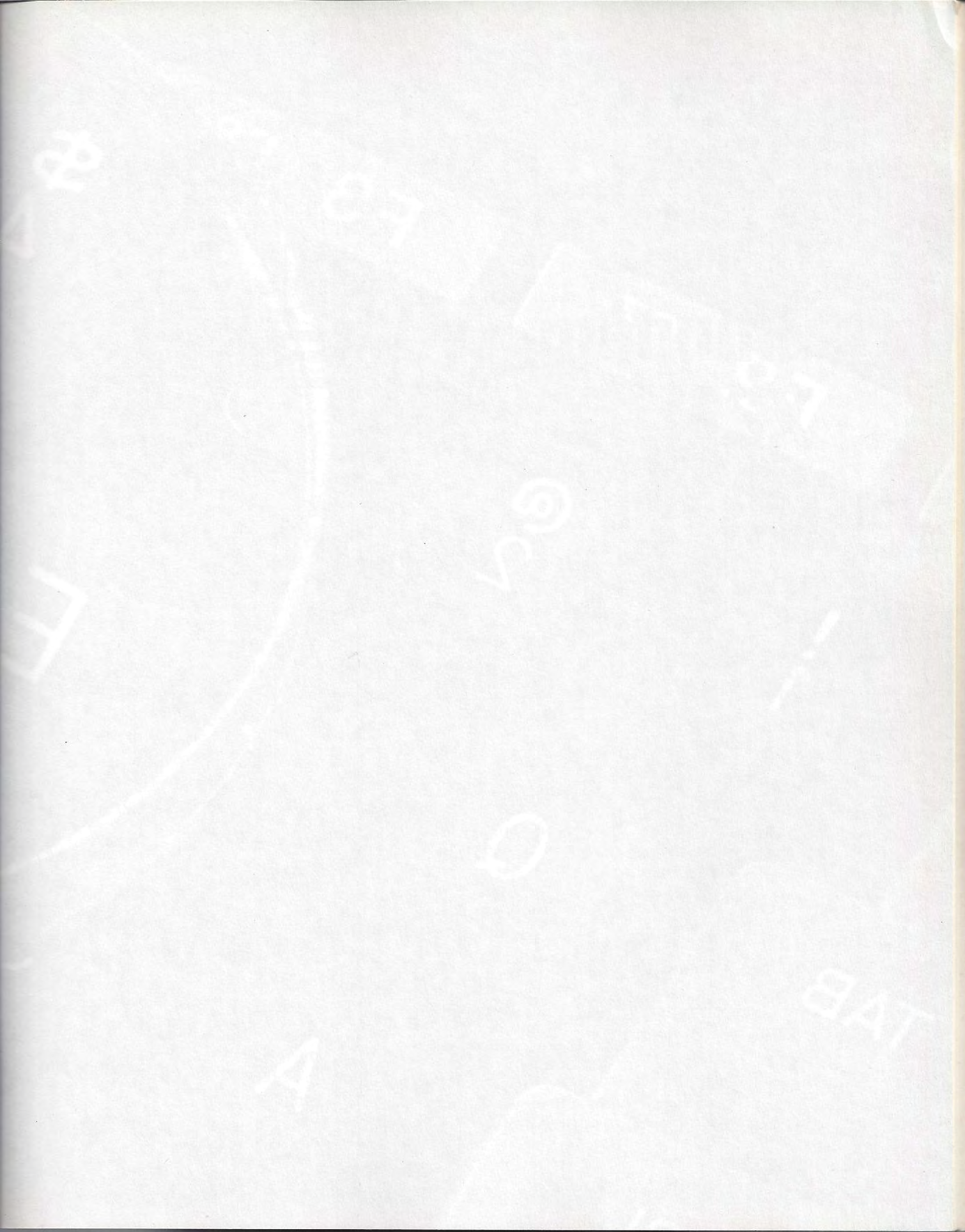
Cuando se escribe un programa, admite dividirse en varias áreas distintas de codificación. A las primeras líneas puede llamárselas el CUERPO del programa; frecuentemente no consisten más que en una serie de llamadas a subrutinas, como se muestra a continuación:

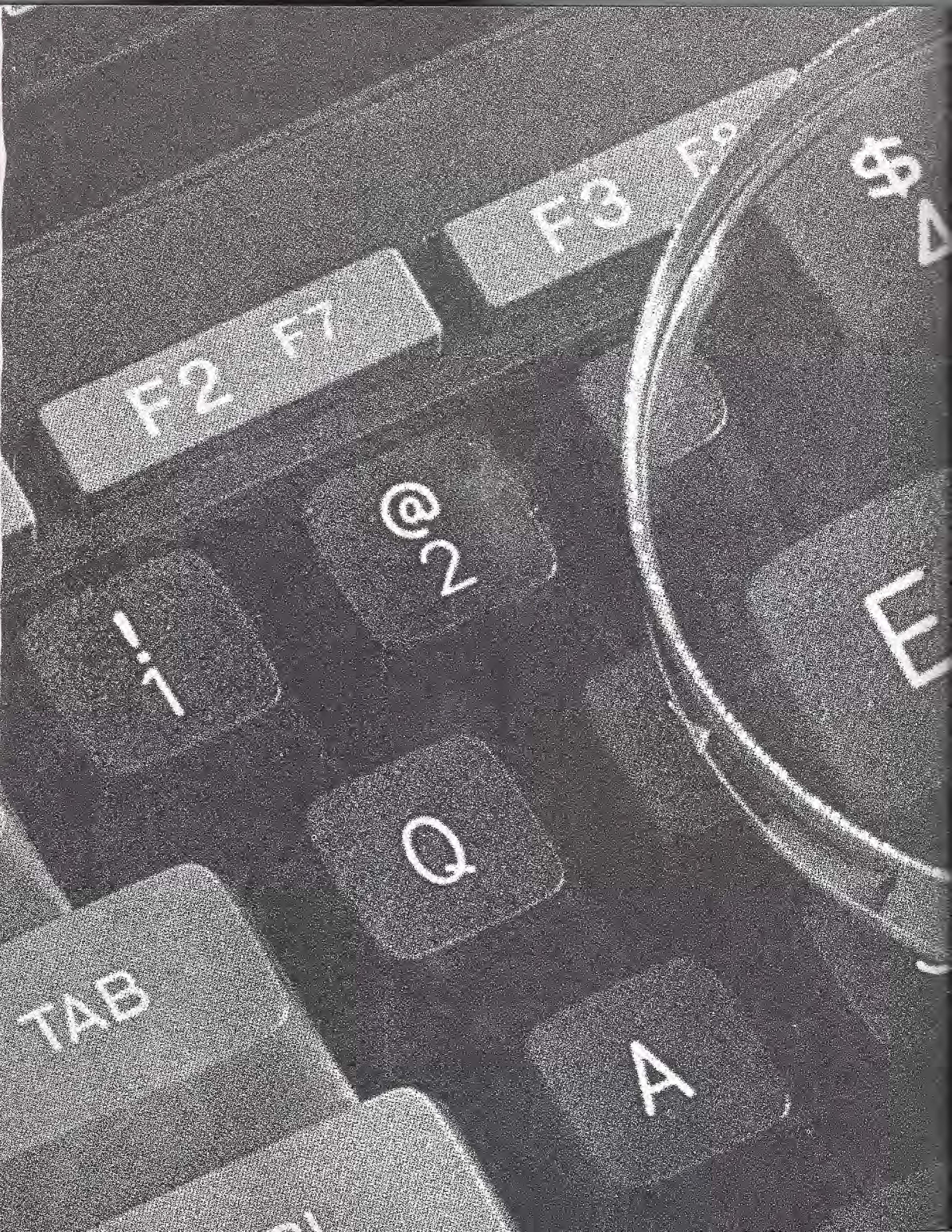
```
10 GOSUB 1000:REM inicializa funciones
20 GOSUB 2000:REM declara variables
30 GOSUB 3000:REM fija pantalla
40 GOSUB 6000:REM juego
50 GOSUB 7000:REM quiere jugar otra vez?
60 IF respuesta$="SI" THEN GOTO 40
70 END
```

El programa continuaría con las definiciones de las subrutinas de la línea 1000 en adelante. Pueden separarse estas dos áreas del programa mediante una línea en blanco, exceptuando un carácter ":". Por tanto,

```
100:
```

deja una línea en blanco dentro del programa. Después de las subrutinas principales, a las que llamamos el cuerpo del programa, definimos las subrutinas utilizadas dentro de las subrutinas principales. Después de las definiciones de subrutinas colocamos las sentencias DATA utilizadas por el programa.




F2 F7
F3
@
2
!
1
Q
TAB
A
E
$



12 Código máquina MSX

La CPU Z-80, el corazón de los ordenadores MSX, está controlada por el programa residente en la ROM MSX. Este programa está escrito de una forma codificada llamada código máquina. Estas instrucciones del código máquina es lo único que puede comprender la CPU Z-80, y cuando tecleamos un programa BASIC y lo ejecutamos, éste se ejecuta como una serie de rutinas en código máquina. Hay, en total, casi setecientas instrucciones diferentes que la CPU Z-80 puede llevar a cabo, y por tanto es obvio que este capítulo no puede hacer más que dar una visión de conjunto acerca de la programación en código máquina del Z-80. Su objetivo, sin embargo, es tratar de aplicar la programación en código máquina del Z-80 a los ordenadores MSX.

Empecemos, por otra parte, sugiriendo un par de libros que tratan detalladamente la programación del Z-80. Son muy indicados *Z-80 Assembly Language Programming* de Lance A. Leventhal, y *Z-80 Microprocessor Programming and Interfacing*, tomo 1, de Nichols, Nichols y Rony. El único problema con este último es que es un libro dirigido a una máquina en particular. No obstante, los principios y explicaciones merecen la pena. Para principiantes se sugiere el segundo título.

Ahora sigamos con el Z-80. El primer registro que vamos a ver es el contador de programas *(Program Counter)*, un registro de 16 bits que permite saber a la CPU qué parte del programa en código máquina se ejecuta en cada momento. Cuando se conecta el ordenador, el PC tiene valor 0. Esto provoca que la CPU ejecute el programa en código máquina que comienza en la dirección de memoria 0, que en el caso de los ordenadores MSX es la ROM. Recordará del capítulo 1 que la

CPU tiene acceso a la memoria del ordenador a través del *bus* de direcciones, y accede al contenido de una posición de memoria dada por medio del *bus* de datos. Cuando se lee una instrucción de la memoria, el Z-80 sabe automáticamente si hay que buscar datos o si el byte es el primero de una instrucción que consiste en varios de ellos, y el PC es actualizado automáticamente para que el Z-80 sea capaz de acceder a la siguiente instrucción una vez ha ejecutado la actual.

Hay otros registros en la CPU, además del PC. Son los expuestos en la figura 12.1, y se llaman "grupo de registros principales" y "grupo de registros alternativos".

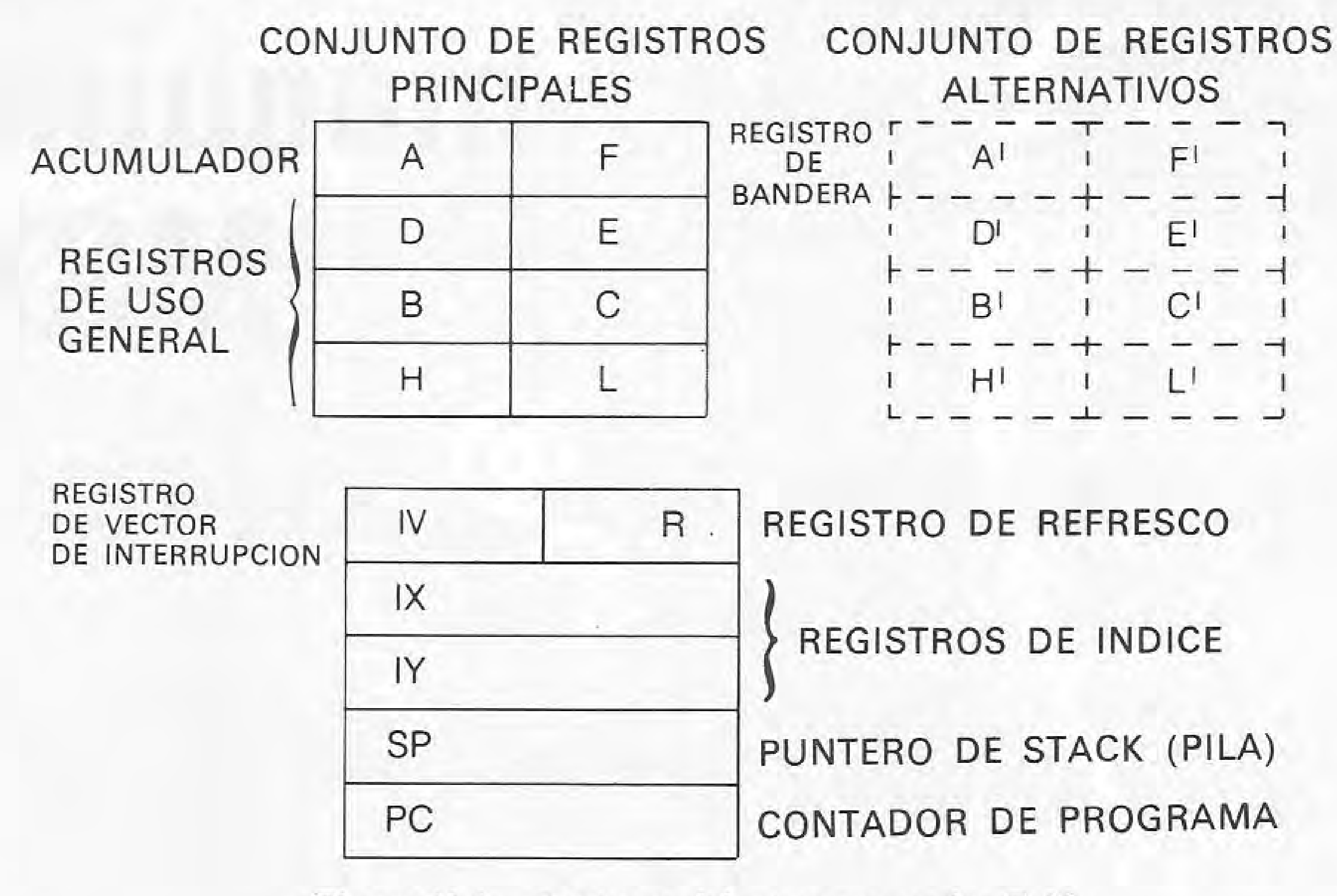


Figura 12.1.—*Registros del microprocesador Z-80.*

Probablemente el más importante de éstos sea el registro A, o acumulador, como se le suele llamar. La CPU realiza muchas de sus operaciones aritméticas con los datos almacenados en el acumulador, y hay un amplio abanico de instrucciones dentro del conjunto de las de la CPU que acceden al acumulador. El acumulador tiene la suficiente longitud para contener números entre 0 y 255, pero los registros BC, DE y HL son cada uno lo bastante grandes como para contener cualquier número entre 0 y 65535. Estos últimos también pueden usarse como registros simples, siendo capaz cada uno de ellos de contener números entre 0 y 255.

El registro F es el "registro de bandera" *(flag register)* de la CPU; realiza un trabajo similar al que hacía el registro de estado del VDP; pero aquí cada bit del registro señala un hecho particular acerca de la CPU. Los valores de los bits del registro de bandera están afectados por operaciones aritméticas, comparaciones entre 2 bytes de datos y otras operaciones.

Los registros IV, R, IY y SP son medianamente especializados, y no los estudiaremos aquí. Cualquier persona que desee hacer un uso completo ee la CPU Z-80 debería tener un libro sobre el tema, estudiarlo y aplicar sus conocimientos al ordenador MSX. El resto de este capítulo tratará de los problemas particulares de

la programación en código máquina para los ordenadores MSX. Se listarán como ejemplo algunas rutinas en código máquina, y se aconseja al lector interesado que las trabaje cuidadosamente sirviéndose de alguno de los libros de referencia mencionados anteriormente.

El primer problema relacionado con la programación en código máquina de los ordenadores MSX es dónde y cómo almacenar y escribir los programas en memoria.

Almacenamiento de programas en código máquina

La manera más indicada de almacenar programas en código máquina en los ordenadores MSX es usar el comando CLEAR para fijar un área de memoria intocable por los programas o variables BASIC.

Por ejemplo, supongamos que quiere reservar 100 bytes de RAM para colocar subrutinas en código máquina. El uso del comando CLEAR le permitirá fijar un área de memoria entre el final del espacio de trabajo del BASIC y el principio del espacio de trabajo del sistema. La manera de reservar memoria de este modo se explica a continuación. Lo primero, es útil conocer dónde termina generalmente el espacio de trabajo del BASIC. En una máquina de 16K es en la dirección 61583. Puede calcular esto haciendo un *reset* en su máquina y evaluando la expresión siguiente:

```
PRINT(comienzo de RAM)+FRE(0)
```

Una vez conocemos esto, para reservar la memoria utilizamos el comando siguiente:

```
CLEAR 200,61482
```

La RAM entre 61483 y 61583 está libre ahora para rutinas en código máquina, que estarán perfectamente a salvo de ser alteradas por el BASIC.

No hay muchos otros lugares factibles para colocar rutinas en código máquina. Otras máquinas pueden almacenar estas rutinas en lugares bastante extraños, tales como una sentencia REM en la primera línea de programa o como contenido de una variable de cadena. Sin embargo, estas técnicas son un poco difíciles de aplicar a los ordenadores MSX.

En el primer caso, la posición de una sentencia REM en la primera línea de programa será la misma para todos los ordenadores que tengan la misma cantidad de memoria. Sin embargo, si cambia la cantidad de memoria poseída por un ordenador, se alteraría la posición en RAM de la sentencia REM.

Respecto a almacenar el programa en una variable de cadena, el problema de calcular la posición en RAM de la cadena cada vez que desea llamar a la rutina en código máquina convierte este método de almacenamiento en insuficiente.

Advierta que no es posible almacenar programas en código máquina en VRAM. Ello sucede por el hecho de que la CPU no puede aceeder directamente a la memoria de video, sino que tiene que hacerlo a través del procesador de *display* de

video. Si lo desea, no obstante, podría almacenar datos usados por programas en código máquina en áreas libres de VRAM. Esto es, sin embargo, un poco arriesgado, ya que al cambiar el modo de pantalla los datos almacenados podrían ser destruidos fácilmente. Asimismo, leer estos datos sería un poco lento, ya que todas las lecturas de la VRAM tendrían que hacerse a través del VDP.

Entrada a programas en código máquina

Como ha podido aprender en cualquier texto de programación en código máquina Z-80, una de estas instrucciones, tal como

```
LD A,23
```

que carga el acumulador con el número 23, se representa en memoria como números, que la CPU reconoce como instrucciones. El comando anterior, por ejemplo, se codifica como los dos números

```
62,23
```

62 es el número que representa la instrucción LD A,*n*, donde *n* es un número entre 0 y 255. 23 es un dato necesario para el comando.

Hay programas disponibles para ordenadores personales que traducen las instrucciones seudo-inglesas como LD A,23 a los números apropiados. Estos programas se llaman ENSAMBLADORES, pero hasta que haya uno disponible para el ordenador MSX[1] tendremos que hacer el trabajo de traducción nosotros mismos. (O, si nos sentimos ambiciosos, ¡podemos escribir el nuestro propio!) La técnica de escribir las instrucciones para el Z-80 en papel y después usar un libro de referencia para obtener los códigos numéricos de cada instrucción se llama ENSAMBLAJE MANUAL, y sólo es realmente apropiado para programas medianamente cortos. Veamos un ejemplo de un programa en código máquina Z-80:

```
LD A,23
LD (52000),A
RET
```

La instrucción RET al final del programa es esencial para asegurar que el control regresa al BASIC después de que el programa ha sido ejecutado por la CPU. Si se omite, probablemente el ordenador se quedará "colgado". Este es el nombre que se da al hecho de que el ordenador sea incapaz de realizar ningún otro proceso. La única manera de recobrar el control es presionar el botón *reset*, pero con esto también perderíamos el programa de la memoria. La secuencia de estos comandos anteriores tiene el mismo efecto que la instrucción BASIC

```
POKE 52000,23
```

El programa siguiente colocará los bytes que configuran el programa en un área particular de memoria. Observe cómo la dirección 52000 se almacena como 2 bytes, siendo colocado el byte menos significativo en primer lugar dentro de la RAM:

```
10 CLEAR 200,61482
20 FOR I=61483 TO 61488
30 READ N:POKE I,N
40 NEXT
50 DATA 62,23:REM codigos para LD A,23
60 DATA 50,32,203:REM codigos para LD (52000),A
70 DATA 201:REM codigo de RET
```

El programa está ahora colocado firmemente en un área segura de memoria. Todo lo que nos queda por hacer ahora es ejecutarlo en código máquina. Esto se hace usando el comando USR.

Uso de DEFUSR y USR

Aunque previamente se han mencionado estos comandos, los veremos ahora de nuevo. Antes de que podamos llamar a cualquier código máquina con USR, debemos informar al ordenador de la dirección del código máquina en cuestión. Supongamos que queremos llamar a nuestro programa con el comando USR1.

Para fijar la dirección a donde queremos llamar al comando USR1, utilizamos una orden DEFUSR. La sintaxis completa de la instrucción es

```
DEFUSRn=entero
```

donde *n* es un dígito entre 0 y 9, y entero es un número entre 0 y 65535, que es la dirección del primer byte del programa en código máquina que va a ser llamado por el comando USR *n*. Como queremos llamar a nuestra rutina con USR1, *n* = 1. La dirección de comienzo del programa en código máquina es 61483, y, por tanto, el comando DEFUSR necesario para decir al ordenador dónde debe ejecutar el programa es

```
DEFUSR 1=61483
```

Esta instrucción debe ejecutarse antes de que intentemos llamar a rutina en código máquina con el correspondiente comando USR. Si quisiéramos usar otra llamada USR, tal como USR2, para llamar a la rutina, simplemente sustituya el 1 por el 2. Observe que si quiere llamar a la rutina con una instrucción USR0 no es necesario el dígito. Así, para llamar al programa con una instrucción USR0,

```
DEFUSR=61483
```

definiría la dirección. Tampoco el número 0 es necesario en la orden USR.

Una vez ha sido asignada la dirección de esta manera, podemos usar el comando USR1 para llamar a la rutina, como se muestra a continuación:

```
PRINT USR1(0)
```

El argumento pasado al comando USR en este caso no tiene ninguna importancia, y es ignorado por la rutina en código máquina. Todo lo que parece ocurrir es que se imprime en la pantalla el valor de dicho argumento. Sin embargo, si ahora usamos el comando PEEK para asegurarnos del valor contenido en la posición 52000 encontraríamos que es 23. La sentencia USR no tiene por qué utilizarse con la orden PRINT; puede usarse como parte de la asignación de una variable. Esto es particularmente útil si la rutina en código máquina retorna un valor al programa BASIC que la llamó.

```
L=USR1(0)
```

es un ejemplo de este método de utilizar USR. Aunque no se usa el argumento en esta aplicación particular, veremos brevemente cómo podemos hacer uso de ello en nuestros programas en código máquina.

Uso de parámetros

Es importante saber pasar parámetros a una rutina en código máquina del usuario. Esto quiere decir que los valores contenidos en las variables BASIC pueden ser empleados por las rutinas en código máquina si es necesario. En los ordenadores MSX el valor pasado entre paréntesis en un comando USR se coloca en un lugar determinado de la memoria. El área de memoria que contiene el parámetro o la información acerca de dónde puede encontrar la rutina el parámetro se llama BLOQUE DE PARAMETRO. Se compone de dos partes. Un solo byte localizado en la dirección 63075 contiene la información relativa al tipo de parámetro pasado a la rutina; o sea, si es entero, real o de cadena. El resto del bloque de parámetro se encuentra entre las direcciones 63478 y 63485. Cuando llama a una rutina USR se coloca un valor en la posición 63075 relacionado con el tipo de variable siguiente.

Observe que el contenido del bloque de parámetro no parece tener mucho sentido si tiene que utilizar para conocerlo el comando BASIC PEEK. Esto puede parecer un impedimento, pero no lo es, ya que accedemos directamente a estas posiciones sólo desde el código máquina.

Tipos de parámetros

El parámetro pasado a una rutina en código máquina para su uso puede ser de cualquier tipo: entero, cadena, simple o doble precisión. Por ejemplo, todas las instrucciones siguientes son perfectamente correctas:

```
L=USR(I%)
L=USR("TEST")
L=USR(3.345#)
```

Los distintos tipos de parámetros se utilizan de la manera siguiente:

Enteros

La posición 63075 contendrá el valor 2. El valor del número se almacena en 2 bytes en las posiciones &HF7F8 y &HF7F9. El valor se almacena con el byte menos significativo en la posición &HF7F8.

Cadenas

Si se pasa como parámetro de la rutina USR una cadena constante o una variable de cadena, la posición 63075 contendrá el valor 3. Las posiciones &HF7F8 y &HF7F9 contendrán la dirección de un DESCRIPTOR DE CADENA, siendo el byte menos significativo de la dirección el de la posición &HF7F8.

El descriptor de cadena es un bloque de memoria de 3 bytes de largo que contiene detalles acerca de la cadena. El primer byte del bloque, señalado por el valor contenido en &HF7F8 y &HF7F9, es la longitud del parámetro de cadena. Los 2 bytes siguientes contienen la dirección de la cadena, con el byte menos significativo primero.

Simple precisión

La posición 63075 contiene el valor 4, y el número de simple precisión está colocado en las posiciones de &HF7F6 a &HF7F9.

Doble precisión

La posición 63075 contiene el valor 8 al entrar en su rutina en código máquina, y el número de doble precisión se coloca en las posiciones de &HF7F6 a &HF7FD.

Veamos ahora una rutina simple en código máquina que acepta un parámetro de BASIC, opera con él y lo devuelve a BASIC. No obstante, antes de hacerlo debemos aprender cómo pasan los valores de la rutina en código máquina a BASIC. Una manera sería dejar el resultado de la operación de la rutina en código máquina en una posición específica de memoria a la que podríamos acceder con el comando PEEK. Un segundo método sería escribir la rutina de tal manera que use el bloque de parámetro para transferir los datos al programa BASIC. Este método es mucho más elegante, y por eso vamos a estudiarlo ahora.

Retorno de valores

Normalmente, a menos que se especifique otra cosa, el valor retornado (de cadena o entero) por una llamada a USR es el mismo que se pasó a la rutina de código máquina en los paréntesis del comando USR. Sin embargo, con las instrucciones apropiadas en la rutina de código máquina es posible retornar un valor calculado por ella.

Enteros

Para devolver un valor entero al BASIC fijamos la posición 63075 al valor 2. Entonces escribimos el valor entero en las posiciones &HF7F8 y &HF7F9, colocando el byte menos significativo del número en la posición &HF7F8.

Cadenas

Fije la posición 63075 al valor 3. Entonces coloque en las posiciones &HF7F8 y &HF7F9 el puntero que señale a un bloque descriptor de cadena situado en algún lugar de la memoria. El bloque descriptor de cadena utilizado aquí tiene la misma estructura que el de pasar parámetros a las rutinas en código máquina.

Simple precisión

Fije la posición 63075 al valor 4, y coloque los bytes que configuran el número en las posiciones de la &HF7F6 a la &HF7F9.

Doble precisión

Coloque a 8 la posición 63075, y los bytes que configuren el número en las posiciones &HF7F6 a &HF7FD.

Al regresar de la rutina en código máquina, el parámetro especificado por cualquiera de los métodos anteriores regresará en lugar del argumento que pasó a la misma. Observe que es posible que se produzcan errores *type mismatch* si configura su programa para obtener un valor de cadena y llama a la rutina en código máquina usando un comando, digamos, como

```
100 LET L=USR(0)
```

Examinemos una rutina que acepta un parámetro pasado en una sentencia USR y retorna un valor entero, después de haber sumado 30 al valor inicial. La rutina espera que se le suministre un valor entero:

```
LD HL,(&HF7F8)  42,248,247
LD DE,30        17,0,30
ADD HL,DE       25
LD (&HF7F8),HL  34,248,247
LD A,2          62,2
LD (63075),A    50,99,246
RET             201
```

La primera línea de este programa carga el entero pasado a la rutina en el comando USR. Las dos siguientes suman 30 a ese número, y las tres que continúan colocan el argumento modificado en la posición adecuada para el retorno a BASIC y fijan la posición 63075 para señalar que el número es un entero. Los números a la derecha del listado son los que representan las instrucciones. Colóquelas en un área de memoria apropiada, y después utilice un comando USR para llamar a la rutina; por ejemplo,

```
DEFUSR1=61483
```

Esta sentencia

```
PRINT USR1(45)
```

retornará el valor 75 y sería visualizado en la pantalla.

Esta capacidad de pasar valores de rutinas en código máquina a programas BASIC es bastante útil, ya que permite al programador escribir rutinas que no serían factibles o serían muy lentas en BASIC, pero aún mantienen el uso de variables BASIC para obtener datos de la rutina en código máquina.

Acceso a otros circuitos integrados

Ya hemos visto cómo podemos desarrollar nuestra técnica de programación del ordenador MSX accediendo directamente a otros *chips* como el generador de sonido programable y el procesador de *display* de video. Ya hemos visto, también, cómo podemos modificar directamente los contenidos de los registros del PSG y del VDP accediendo al mapa IN/OUT del ordenador utilizando las instrucciones BASIC INP y OUT. Esto se estudió en el capítulo 10, donde también vimos cómo modificar directamente los contenidos de la memoria de video usando el VDP.

No es sorprendente, pues, que podamos realizar una tarea similar empleando los comandos Z-80 que utilizan directamente el mapa IN/OUT de la CPU. Cualquier persona que desee usar estos *chips* desde el código máquina debe leer el capítulo correspondiente, ya que las técnicas para controlar el comportamiento del VDP, alterando el contenido de sus registros, será el mismo, lo hagamos en BASIC o en código máquina.

No encontramos problemas particulares al modificar valores del PSG o del PPI. El comando

```
OUT n,A
```

del código máquina Z-80 escribe el contenido del acumulador en la posición I/O *n*. Pueden encontrarse más detalles acerca del comando en cualquiera de los trabajos de referencia mencionados al principio del capítulo. Para leer un valor de una posición de INPUT/OUTPUT concreta, el comando

```
IN A,n
```

puede utilizarse; leerá un valor de la posición I/O *n* y lo colocará en el acumulador. Los métodos empleados para leer y modificar los puertos de I/O de cada *chip* serán los mismos que desarrollamos en el capítulo 10.

Cuando llegamos a acceder al VDP desde el código máquina deben seguirse las líneas de referencia mencionadas en el capítulo 10, especialmente en lo que respecta a la lectura del registro de estado antes de modificar los otros registros. El único problema al acceder al VDP usando el código máquina se refiere a cuestiones de sincronización. No podemos modificar los valores de los registros del VDP a una frecuencia mayor de 1 cada 4,3 milisegundos. Por tanto, puede ser necesario colocar bucles de demora en sus programas en código máquina para asegurarse de que la operación no se desarrolla a demasiada velocidad.

Aparte de esto, puede accederse al VDP de manera análoga a como lo hicimos en el capítulo 10. Es precisamente en el acceso a la VRAM desde el código máquina donde el autoincremento tiene gran valor; una vez se ha indicado al VDP una dirección, pueden utilizarse instrucciones OUT repetidas para enviar otros bytes a la VRAM, siendo incrementada cada vez la posición donde se escribe el byte de datos. Esto hace más fácil la transferencia de datos entre la CPU y la VRAM, ya que solamente necesitamos especificar la dirección una vez. Esto, evidentemente, no es útil si las posiciones a modificar en la RAM de video no son contiguas.

Anzuelos

No, no tiene nada que ver con la pesca. Un ANZUELO es un medio por el cual el programador MSX puede modificar el comportamiento del ordenador cuando está rodando una rutina de la ROM MSX. Esto es factible en virtud del hecho de que varias rutinas de la ROM, en algún punto de su ejecución, llaman a una posición de RAM. Por ejemplo, la rutina que se ejecuta cada vez que hay una interrupción del VDP llama a una rutina en RAM en la dirección &HFD9A. Si observamos esta posición, encontramos que contiene una sentencia RET y por eso, normalmente, el llamar a esta dirección no tiene ningún efecto. Esta posición, y los 4 bytes siguientes, conforman un BLOQUE DE SERVICIO, y al conectar o al hacer un *reset* en su ordenador contiene el valor 201, que es el código de RET. La manera en que modificamos el comportamiento de la rutina ROM es colocar una instrucción CALL o JP apropiada, con la dirección a la que queremos que salte la rutina ROM. Seguidamente se muestra un ejemplo:

```
CALL 52000
RET
RET
```

El comando CALL ocupa los primeros 3 bytes del bloque de servicio, y las dos sentencias RET siguientes todavía ocupan los bytes sin utilizar del bloque.

Esta rutina provocará, cada vez que ocurra una interrupción, un salto a la subrutina de la posición 52000. Si prueba esto, escriba la rutina de la posición 52000, o dondequiera que decida poner su rutina de servicio, antes de cambiar el bloque de servicio para llamarla. Si no lo hace, el resultado será que el ordenador quedará "colgado" casi inmediatamente.

Con lo expuesto completamos la revisión de conocimientos especiales que son útiles cuando programamos los ordenadores MSX en código máquina. Utilice los libros citados para desarrollar sus habilidades como programador en código máquina, y practique escribiendo rutinas sencillas al principio. No sea ambicioso; el BASIC MSX es una implementación extremadamente poderosa del lenguaje, y muchas veces puede resultar completa sin el recurso del código máquina.

NOTAS DEL CAPITULO

1.—En la actualidad existe un ensamblador-desensamblador para MSX llamado DEVPAC, que ha sido desarrollado y comercializado por:

HISOFT
180 High Street North
Dunstable, Beds. LU 6 1AT
INGLATERRA

a un precio aproximado de 20 libras.

del código máquina Z-80 escribe el contenido del acumulador en la posición I/O n. Pueden encontrarse más detalles acerca del comando en cualquiera de los trabajos [illegible] de INPUT, OUT [illegible]

Los métodos empleados para leer y modificar [illegible] sincronización. No podemos [illegible] frecuencia mayor de 1 cada 43 milisegundos. Por tanto, puede ser necesario colocar bucles de demora en sus programas en código máquina para asegurarse de que la operación no se desarrolle a excesiva velocidad.

Aparte de esto, puede accederse al VDP de manera análoga a como lo hicimos en el capítulo 30. Es precisamente en el acceso a la VRAM desde el código máquina donde el autoincremento tiene gran valor; una vez se ha indicado al VDP una dirección, pueden utilizarse instrucciones OUT repetidas para enviar muchos bytes a la VRAM, siendo incrementada cada vez la posición donde se escribe el byte de datos. Esto hace más fácil la transferencia de datos entre la CPU y la VRAM, ya que solamente necesitamos especificar la dirección una vez. Esto, evidentemente, no es útil si las posiciones a modificar en la RAM de vídeo no son contiguas.

Anzuelos

No, no tiene nada que ver con la pesca. Un ANZUELO es un medio por el cual el programador MSX puede modificar el comportamiento del ordenador cuando está recorriendo una rutina de la ROM MSX. Esto es factible en virtud del hecho de que varias rutinas de la ROM, en algún punto de su ejecución, llaman a una posición de RAM. Por ejemplo, la rutina que se ejecuta cada vez que hay una interrupción del VDP llama a una rutina en RAM en la dirección &HFD9A. Si observamos esta posición, encontraremos que contiene una sentencia RET [illegible] el llamar a esta dirección no tiene ningún efecto. Esta posición, y los 4 bytes siguientes, conforman un BLOQUE DE SERVICIO, y al comenzar [illegible] en su ordenador contiene el valor 201, que es el código de RET. [illegible] que modificamos el comportamiento de la rutina ROM es colocar una [illegible] apropiada, con la dirección a la que queremos que salte la rutina ROM. Seguidamente se muestra un ejemplo.



Apéndice Sistemas de numeración

El acto de contar es automático para la mayoría de nosotros, pero cuando escribimos el número 234, ¿qué queremos decir exactamente? Para ser capaces de comprender este método de escribir números es necesario examinar primero la totalidad del proceso numérico; un proceso que es casi intuitivo para nosotros.

Lo primero a tener en cuenta es que el número se escribe en columnas, de izquierda a derecha del papel. Estas columnas no tienen todas la misma significación respecto al valor del número. Examinemos un número cualquiera: 234. Decimos que la columna más a la derecha del número se llama columna de las UNIDADES, y en esta columna contamos lo que se llaman DIGITOS. En nuestro sistema de numeración, basado en 10 dígitos y llamado, por ello, sistema decimal o DENARIO, los dígitos son 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9.

En cualquier sistema de numeración el número de dígitos disponible tiene un nombre especial: la RAIZ o BASE del sistema. Por eso, con los 10 dígitos del sistema decimal, tenemos una base 10, ya que pueden aparecer 10 dígitos diferentes en la columna de las unidades. El proceso de contar las unidades es, sencillamente, ir pasando sucesivamente de uno a otro de los dígitos disponibles en el orden correcto. Por eso, para el sistema decimal pasamos uno a uno los dígitos de 0 a 9 en esa secuencia. Sin embargo, ¿qué ocurre cuando hemos pasado todos?

En términos más específicos, ¿qué ocurre cuando llegamos a 9 en base 10, y tenemos que añadir una unidad más? Pues se pone la columna de las unidades a cero, y colocamos el siguiente número en la siguiente columna a la izquierda. La operación que acabamos de hacer se llama ACARREO. Cada dígito de la columna a la izquierda de la columna de las unidades difiere del anterior en esa misma columna en la base del sistema de numeración. En el sistema decimal la base es 10, y esta co-

lumna se llama, por ello, columna de las decenas. Entonces prosigue la cuenta en la columna de las unidades hasta que se genere el siguiente acarreo y se añade un dígito a la columna de las decenas. De manera similar, una vez alcanzamos 9 en la columna de las decenas, se lleva a cabo un acarreo en la siguiente columna a la izquierda, en el que las unidades difieren una de otra en la base multiplicada por sí misma. En el sistema decimal es igual a 100 y por eso llamamos, a la columna a la izquierda de la columna de las decenas, la columna de las centenas. Si aplicamos esta información al número 234 vemos que tiene 4 unidades, 3 decenas y 2 centenas. Cada unidad en la columna de las unidades incrementa el valor del número en uno; en la columna de las decenas, en diez, y en la de las centenas, en cien.

No obstante, hay otros sistemas de numeración además del decimal. Los babilonios utilizaban un sistema en base 60, de donde derivan nuestros métodos de medida de ángulos y tiempo[1]. Hay quien todavía emplea un sistema en base 12 para medir pies y pulgadas, siendo 1 pie = 12 pulgadas. Este sistema de medida también utiliza la base 3, ya que hay 3 pies en una yarda. En cualquiera de estos sistemas de numeración, sin importar la base, cada unidad en la columna de la izquierda incrementa el valor del número por la base del sistema de numeración respecto a la columna anterior.

Si los ordenadores trabajaran en base 10, no habría razón para preocuparnos acerca de otras bases numéricas distintas de la decimal. Sin embargo, debido al hecho de que es más fácil construir circuitos electrónicos que detecten la presencia o ausencia de una señal que circuitos capaces de diferenciar 10 niveles de señal diferentes, los ordenadores trabajan en base 2. Su ordenador MSX realmente reconoce sólo dos dígitos, el 0 y el 1, y reconoce estos como niveles de voltaje diferentes en sus circuitos internos. Estos dígitos se representan por una señal de 5 voltios para el dígito 1 y una señal de 0 voltios para el 0. Este sistema de numeración en base 2 se llama "sistema BINARIO". Por eso, provistos de los dígitos 0 y 1, ¿cómo contamos en binario?

Bien. En lugar de las unidades, las decenas y las centenas de que consta el sistema decimal, en el binario tenemos unidades, doses y cuatros. Cuando tenemos un 1 en una columna y hemos de añadir uno a la cuenta, generamos un acarreo a la siguiente columna y reemplazamos el 1 por un 0. El acarreo va hacia la columna a la izquierda, a la que se le ha añadido una unidad. Es decir, el proceso de contar en binario es enteramente análogo al de hacerlo en decimal.

Como regla general en los sistemas de numeración decimos que cualquier columna es menos significativa respecto al valor del número que respecto a sí misma. Es decir, una unidad en la columna de las unidades añade menos valor al número que el que añadiría si estuviera en la columna de las decenas.

En todos los sistemas de numeración, la columna de las unidades contiene lo que se llama el DIGITO MENOS SIGNIFICATIVO y la columna más a la izquierda contiene el DIGITO MAS SIGNIFICATIVO del número. La tabla muestra los efectos de contar en binario (los subíndices indican la raíz o base):

1_{10}	0001_2
2_{10}	0010_2
3_{10}	0011_2
4_{10}	0100_2

Observe que los ceros de la izquierda no influyen en el valor del número. Si analizamos uno de estos números binarios, digamos 0011, podemos ver que hay una unidad y un dos. Esto nos proporciona el valor de 3 para el número en decimal. Si añadimos una unidad más, tendremos que hacer un acarreo desde la columna de las unidades a la de los doses colocando un 0 en la columna de las unidades. Sin embargo, la columna de los doses tiene ya un 1, y por eso debemos hacer otro acarreo de la columna de los doses a la de los cuatros, colocando un 0 en la columna de los doses. Así, obtendríamos 0100, o 4 en decimal. Si descartamos el primer 0, no influye de ningún modo en el valor del número; vemos que para representar el número 4 en binario necesitamos tres dígitos, mientras que en decimal sólo necesitamos uno. En el sistema decimal el número más grande que podemos representar con tres dígitos es el 999, mientras que en binario sólo podremos representar 111 ó 7 en decimal.

Esto nos da una idea general acerca de los sistemas de numeración; según decrece el número de dígitos disponible, también lo hace el valor del número mayor que podemos representar con un número de dígitos dado. Al contrario, aumentando el número de dígitos disponibles podemos aumentar el valor del mayor número que podemos representar con ese número de dígitos. Con esto vemos que la raíz es el número de dígitos diferentes de un sistema de numeración. El valor del número mayor que se puede conseguir con un número dado de dígitos depende de la base del sistema de numeración.

Hasta ahora hemos desarrollado la manera de representar un número y presentado someramente el sistema binario. Como nuestros ordenadores trabajan en binario, es el momento de estudiar más detenidamente el sistema binario y aprender a realizar operaciones aritméticas sencillas en este sistema de numeración.

Aritmética binaria

La operación artimética más simple en el sistema decimal es la adición, que puede verse como una extensión de los procesos que hemos visto al contar. Veremos el proceso de suma binaria examinando primero la suma de 1 a 0010. Esta suma es una aplicación directa del conteo visto anteriormente, pero con él veremos el proceso normal para realizar una suma binaria. En primer lugar colocamos los números de tal manera que los números menos significativos estén en la misma columna, como se expone a continuación:

$$\begin{array}{r} 0010 \\ \underline{0001} \end{array}$$

Si encontramos un solo número, como 1 en el ejemplo anterior, suponemos que es una unidad, a menos que se diga otra cosa específicamente. De manera similar, en el número binario 10 se supondría que el 0 está en la columna de las unidades, y el 1 en la columna de los doses. Cuando llevamos a cabo una suma en cualquier sistema

de numeración, siempre comenzamos con el número menos significativo y trabajamos de izquierda a derecha.

Así, en el ejemplo anterior, empezamos sumando el 1 al 0 en la columna de las unidades. Esto nos da el valor de 1 en la columna de las unidades, y no tenemos que hacer ningún acarreo. Estaríamos en la situación siguiente:

```
 0010
 0001
-----
    1
```

Ahora sumamos los dígitos de la columna de los doses. En este caso sumamos 1 y 0. Si no hubiera alguno de los números implicados en el problema de adición en la columna de los doses, sencillamente suponemos que es 0, como hemos hecho en este ejemplo. Otra vez tenemos que colocar un valor de 1 en la columna de los doses sin tener acarreo, obteniendo la situación siguiente:

```
 0010
 0001
-----
 0011
```

Ahora podríamos seguir sumando los dígitos en la columna de los cuatros, pero no es el caso, ya que el resto de los dígitos son ceros.

Por, tanto el resultado en binario es 0011. Examinemos algo más de los números binarios, e intentaremos hacer más sumas. Advierta el hecho de que necesitamos cuatro dígitos para representar números binarios entre 8 y 15. La nueva columna que usamos es la columna de los ochos.

Decimal	**Binario**
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Consideremos la adición de 0110 a 0111. En este problema podemos examinar qué ocurre en la suma cuando se genera un acarreo.

```
         0110
         0111
Acarreo
         ----
         0001
```

Anteriormente podíamos ver el resultado de la suma de las columnas de las unidades. No ha sido necesario el acarreo. Prosigamos con la columna de los doses.

```
         0110
         0111
Acarreo   1
         ----
         0001
```

Es necesario el acarreo y se coloca un cero en la columna de los doses. Vayamos ahora con la columna de los cuatros. Primeramente ignoremos el acarreo de la columna de los doses y sumemos los dígitos de la columna de los cuatros. Tenemos 1 + 1 = 0 acarreo 1. Ahora sumemos el acarreo. Tendremos 1, del acarreo de la columna de los doses, y 0, que obtuvimos al sumar la columna de los cuatros. Luego tenemos el siguiente resultado:

```
         0110
         0111
Acarreo  11
         ----
         0101
```

Vamos ahora a la columna de los ochos. Aunque los números de ambos dígitos en esta columna son ceros, todavía tenemos que considerar el acarreo generado en la adición de la columna de los cuatros. Simplemente añadimos el acarreo a cero:

```
         0110
         0111
Acarreo  11
         ----
         1101
```

Con esto completamos la suma. Si intentamos sumar 1111 y 0001 tendremos series de dígitos de acarreo:

```
         1111
         0001
Acarreo     1
         ----
         0000
```

Ahora tenemos:

```
           1111
           0001
Acarreo    11
          ------
           0000
```

Estos resultados eventuales en el acarreo van a la columna quinta a la izquierda, que en binario es la columna de los dieciséis. Aunque tenemos solamente cuatro dígitos en los dos números que estamos sumando, suponemos la existencia de ceros a la izquierda en la columna de los dieciséis en ambos números:

```
          01111
          00001
Acarreo   1111
         -------
          10000
```

Por tanto, el resultado de esta suma es 16, que no tiene unidades, doses, cuatros u ochos, sino un dieciséis. Esto indica que podemos averiguar por un sencillo examen el valor más alto de un número que se puede representar con un número de dígitos dado.

Con cuatro dígitos binarios, el número más alto que se puede representar es 15; con 3 bits, el más alto es 7. Inténtelo usted mismo. Con cinco dígitos el número más alto es aquel en el que todos sus bits son 1. Esto daría:

$$(1*16)+(1*8)+(1*4)+(1*2)+(1*1)=31$$

El número mayor que podemos representar con un número binario de 5 dígitos es, por tanto, 31. Con un examen más riguroso observamos que en el sistema binario el número más alto que se puede representar con n dígitos es

$$2^n - 1$$

Es decir, el número más alto que podemos representar es 2 multiplicado por sí mismo n veces menos 1. Como comprobación de esta ecuación general sustituyamos algunos valores para observar los resultados. Para $n = 2$, 2 multiplicado por sí mismo 2 veces es 4. Luego, el mayor número que se puede representar con 2 dígitos es $4-1$ ó 3. Para comprobarlo contemos los números posibles con 2 bits, escribiéndolos mientras lo hacemos:

Binario	**Decimal**
00	**0**
01	**1**
10	**2**
11	**3**

Para seguir contando, o sea, para añadir otra unidad, tendríamos que hacer un acarreo a la columna de los cuatros, y necesitaríamos un tercer dígito. Así, la regla se cumple. Esta sencilla ecuación resulta ser cierta para todos los sistemas de numeración, y podemos generalizarla así:

$$X = R^n - 1$$

donde X es el número más alto que se puede representar con *n* dígitos, y R es la base del sistema de numeración. Si comprobamos esto con dos dígitos en base 10 tenemos un valor máximo de (10*10)−1 o 99, que es correcto. Para seguir adelante, es decir, para representar el número 100, necesitamos tres dígitos.

Antes de seguir adelante con la aritmética binaria es necesario definir algo de terminología usada comúnmente en este campo, y que seguramente se encontrará en la programación en código máquina. La frase dígito binario ha sido sustituida en el uso común con la palabra BIT, que viene del inglés *Binary digIT*.

Igual que hemos visto los dígitos más y menos significativos de un número en cualquier sistema de numeración, podemos tener los bits más y menos significativos de un número binario. Cuando hablamos de bits, realmente no es necesario resaltar que la base del sistema en el que estamos trabajando tiene que ser binaria. El bit menos significativo de un número binario se suele llamar LSB[2] *(Least Significant Bit)*. Análogamente, se llama MSB[3] al bit más significativo *(Most Significant Bit)*. La longitud de un número binario según el número de dígitos se da en bits. Así, un número de 4 bits tiene los mismos dígitos. Como el valor más alto que puede representar un número binario depende del número de bits disponibles, un número de 8 bits puede alcanzar un valor máximo más alto que el que pueda alcanzar uno de 4 bits.

Después de este inciso regresemos al tema de la aritmética. En el sistema decimal los números pueden tomar valores entre 0 y 1, y entre otros dos números enteros cualesquiera. Algunos ejemplos son 0.5, 4.7 y 234.567. Estos números se llaman REALES, mientras que los otros son números ENTEROS. En el sistema decimal llamamos al "." del número PUNTO DECIMAL. El número inmediatamente a su izquierda está en la columna de las unidades, y los que están a la derecha son la parte FRACCIONARIA del número. En el sistema decimal la primera columna fraccionaria se llama columna de las décimas; la siguiente, de las centésimas, y se sigue la misma denominación con todas las columnas. Por eso, el valor de un dígito en el número decrece según vamos del punto decimal hacia la derecha. En los números con parte fraccionaria el número menos significativo es el que está en la columna más a la derecha, en lugar del que estaba en la columna de las unidades.

243.567

Así, en el número anterior, el 7 es el dígito menos significativo del número, y representa 7 milésimas del valor de una unidad. También es posible representar fracciones en el sistema binario, donde tenemos un punto binario en vez de un punto decimal. En un número binario con parte fraccionaria la columna a la derecha del punto es la columna de las mitades, la siguiente a la derecha es la columna de los cuartos, etc. Sumamos fracciones binarias exactamente igual que lo hacíamos con

números binarios enteros, empezando en la columna de la derecha y siguiendo hacia la izquierda, llevando los bits de acarreo que sean necesarios. De igual forma podemos llevar un bit de acarreo de la columna de las mitades a la de las unidades cruzando el punto binario si es necesario.

Tras la adición, la operación aritmética más comúnmente usada es la sustracción. El algo más complicada que la suma de dos números, y por eso la veremos con más detalle. En realidad es la operación inversa de la adición. Si examinamos las sumas binarias simples podemos esbozar algunas reglas para restar números binarios:

$$1 + 1 = 10 \quad \text{entonces} \quad 10 - 1 = 1$$
$$1 + 0 = 1 \quad \text{entonces} \quad 1 - 1 = 0$$
$$0 + 0 = 0 \quad \text{entonces} \quad 0 - 0 = 0$$

Los problemas empiezan cuando consideramos la sustracción de 1 a 0. La manera de solventar este problema aparente es "pedir prestado" un dígito a la siguiente columna más significativa del número. El valor del dígito prestado es el que tenía en su columna original; por tanto, si en la columna de los doses tomamos un dígito de la de los cuatros, el dígito tendría un valor de 10 y no de 1. Este es un proceso análogo al que realizamos en la sustracción decimal. Se trata del proceso de acarreo pero en sentido inverso, aunque le llamaremos del mismo modo. Para ver un ejemplo más concreto hagámoslo con una resta binaria:

```
          1101
         -0101
Acarreo
         -----
```

Como siempre, empezamos en la columna de las unidades, con la resta de uno a uno. Esto nos da 0.

```
          1101
         -0101
Acarreo      0
         -----
          0000
```

En la columna de los doses tenemos la situación siguiente (otra vez, no necesitamos realizar el proceso de acarreo para seguir con la sustracción):

```
          1101
         -0101
Acarreo     00
         -----
          0000
```

Ahora podemos calcular el resultado de las columnas de los cuatros y de los ochos, y se observa que en este problema no hemos necesitado realizar el proceso explicado anteriormente. El resultado es el siguiente:

$$
\begin{array}{rr}
 & 1101 \\
 & -0101 \\
\text{Acarreo} & \underline{0000} \\
 & 1000
\end{array}
$$

Debemos ver ahora un ejemplo de sustracción con acarreo en decimal antes de poder atacar el problema en binario. En una resta como 432 menos 56 se enseña a los niños en la escuela a descomponer los dos números de la manera siguiente:

$$
\begin{array}{r}
432 \\
-56 \\
\hline
\end{array}
\quad \text{equivale a} \quad
\begin{array}{r}
400 + 30 + 2 \\
-[\qquad 50 + 6] \\
\hline
\end{array}
$$

La resta se lleva a cabo realizando un acarreo de la columna de las decenas a la de las unidades:

$$
\begin{array}{r}
400 + 20 + 12 \\
-[\quad + 50 + \ \ 6] \\
\hline
\end{array}
$$

Vemos en la columna de las decenas que 50 es mayor que 20; por tanto, se necesita otro acarreo desde la columna de las centenas:

$$
\begin{array}{r}
300 + 120 + 12 \\
-[\qquad 50 + \ \ 6] \\
\hline
\end{array}
$$

El problema es fácilmente resoluble ahora...

$$
\begin{array}{r}
300 + 120 + 12 \\
-[\qquad 50 + \ \ 6] \\
\hline
300 + 70 + 6
\end{array}
$$

... para dar el resultado de 376.

Se puede usar una aproximación similar en la siguiente sustracción binaria:

$$
\begin{array}{r}
1100 \\
-0101 \\
\hline
0111
\end{array}
\quad \text{es equivalente a} \quad
\begin{array}{r}
12 \\
5 \\
\hline
7
\end{array}
$$

Procedamos ahora a descomponer los dos números de la resta binaria:

$$\begin{array}{r} 1100 \\ -0101 \\ \hline \end{array} \quad \text{es equivalente a} \quad \begin{array}{r} 1000 + 100 + 0 + 0 \\ -[\quad 0 + 100 + 0 + 1] \\ \hline \end{array}$$

Como en el ejemplo anterior, es necesario hacer acarreos hasta que los números de la fila inferior sean menores que los correspondientes de la fila superior.

Se hace un acarreo desde la columna de los doses a la de las unidades; como esta columna de la fila superior ya es 0, tendremos un valor de -10;

$$\begin{array}{r} 1000 + 100 + (-10) + 10 \\ -[\quad 0 + 100 + \quad 0 + \ 1] \\ \hline \end{array}$$

Asimismo se lleva a cabo un acarreo de la columna de los cuatros a la de los doses para eliminar el valor -10:

$$\begin{array}{r} 1000 + \quad 0 + 10 + 10 \\ -[\quad 0 + 100 + \ 0 + \ 1] \\ \hline \end{array}$$

Para tener un valor mayor que 0 en la columna de los cuatros se hace un acarreo desde la columna de los ochos:

$$\begin{array}{r} 0 + 1000 + 10 + 10 \\ -[0 + \ 100 + \ 0 + \ 1] \\ \hline \end{array}$$

Se realiza ahora la sustracción...

$$\begin{array}{r} 0 + 1000 + 10 + 10 \\ -[0 + \ 100 + \ 0 + \ 1] \\ \hline 0 + \ 100 + 10 + \ 1 \end{array}$$

... para dar un resultado de $100 + 10 + 1$, que es equivalente a 111 o, en decimal, 7, como queríamos demostrar. Este método de sustracción binaria es igual de laborioso para los ordenadores que para los humanos; pero ¿hay otro camino más fácil?

En la aritmética decimal la sustracción puede generar frecuentemente números menores que 0. Se llaman NUMEROS NEGATIVOS, y en decimal van precedidos por un signo "$-$". Aparecen cuando restamos a un número pequeño otro mayor. También podemos representar números negativos en la aritmética binaria, y lo hacemos usando el COMPLEMENTO A 2.

Antes de explicar de dónde proviene este nombre, veamos cómo podemos representar números positivos (es decir, con un valor mayor que 0) y negativos de 4 bits en complemento a 2. (Es esencial indicar el número de bits con el que vamos a trabajar cuando empleamos el complemento a 2 por una razón obvia que veremos en seguida.) Fíjese qué ocurre cuando pasamos de 0 en los números negativos:

Decimal	Complemento a 2
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
−1	1111
−2	1110
−3	1101
−4	1100
−5	1011
−6	1010
−7	1001
−8	1000

Advertimos varias cosas rápidamente. Con 4 bits normalmente podemos representar números de 0 a 15, es decir, 16 números distintos, siendo el 15 el valor más alto que podemos conseguir con un número de 4 bits. Aquí, sin embargo, todavía tenemos 16 números diferentes, pero el valor más alto que se puede representar es 7. La mitad de los códigos representan números enteros positivos y 0, y la otra mitad representa números enteros negativos. La parte del número que dice si el número es positivo o negativo es el BMS (MSB). En el ejemplo precedente es 0 para un número positivo y 1 para uno negativo. Si tuviéramos un número de 8 bits en complemento a 2 el BMS también indicaría si el número era positivo o negativo, exactamente de la misma manera.

Respecto al rango de números que se pueden representar con un número de bits dado, podemos averiguarlo gracias a la ecuación siguiente:

$$-2^{n-1} < X < 2^{n-1} - 1$$

Con esta ecuación obtenemos el rango de números que podemos representar con la notación de complemento a 2 y con n bits. Por tanto, si $n = 4$ vemos que el rango es DESDE −8 A 7. Que es lo que encontramos en el ejemplo anterior. Así, si hemos dicho que un número está en formato de complemento a 2, podemos decir si es positivo o negativo observando simplemente el BMS. Debemos observar que la notación de un número positivo en complemento a 2 es la misma que cuando se representa en notación binaria normal.

Ser capaces de representar los números de esta manera es extremadamente importante. Primero, nos permite exponer números menores que 0, y, segundo, nos permite reducir el proceso de sustracción, con toda su complejidad inherente, a una adición. Esta última característica es muy importante para nosotros en el campo de los microordenadores, ya que para los circuitos del ordenador es más fácil llevar a cabo una suma que una resta. Veamos cómo funciona. Examinando la lista de los

números en complemento a 2 dada anteriormente, podemos ver que tenemos números positivos y negativos. En la artimética decimal, si sumanos un número y su equivalente negativo el resultado es cero.

¿Qué ocurre si intentamos hacer esto con los números en complemento a 2? Probémoslo con 1 y −1 en la suma de ellos con 4 bits en complemento a 2:

$$\begin{array}{rr} & 0001 \\ + & 1111 \\ \hline & 1\ 0000 \end{array} \qquad \begin{array}{rr} & 1 \\ + & (-1) \\ \hline & 0 \end{array}$$

A primera vista nuestro resultado de 1 0000 ciertamente NO es igual a 0, y sabemos por experiencia que 1 + (−1) ineludiblemente ES 0. Lo que tenemos que apreciar es que tenemos un número de 5 bits. ¿Recuerda que se ha dicho antes que en la aritmética en complemento a 2 es muy importante el número de bits? Bien, aquí es donde esta importancia llega a ser aparente. Empezamos con dos números de 4 bits, y por tanto el resultado también tiene que ser un número de 4 bits en complemento a 2. Si nos deshacemos del BMS, nuestro resultado anterior realmente es 0. Por eso, cuando usamos la aritmética en complemento a 2, hacemos que la sustracción sea tan fácil de realizar como la adición. La desventaja es, desde luego, que necesita más bits para representar un número positivo dado en complemento a 2 que en notación normal. Sin embargo, esto es despreciable frente a las ventajas que nos proporciona la notación en complemento a 2.

Por tanto, ¿cómo convertimos un número binario a la notación en complemento a 2? Es una operación bastante sencilla. El primer paso es el proceso de COMPLEMENTACION, término usado para designar la tarea de sustituir cada 1 del número por un 0, y cada 0 por un 1. Así, el número 1010 complementa al 0101. El segundo paso de la operación sencillamente es sumar uno al número obtenido en el anterior proceso.

Para conseguir el complemento a 2 de 0010, realizamos las siguientes operaciones:

1. Complemento del número; tendremos 1101.
2. Sumar 0001 al número; tendremos 1110.

Eso es todo lo que hay que hacer para obtener el complemento a 2 de un número binario. Una vez hecho, podemos realizar la sustracción muy fácilmente.

Los pasos a seguir para llevar a cabo una sustracción en complemento a 2 de dos números son los que vienen a continuación:

1. Calcular el complemento a 2 del primer número.
2. Calcular el complemento a 2 del segundo número.
3. Sumar los dos complementos a 2.
4. Descartar el BMS (MSB).

¿Qué ocurre cuando intentamos calcular el complemento a 2 en 4 bits de un valor de 1000? Si seguimos las instrucciones precedentes tendremos el número 1000.

Esto no puede ser correcto, ya que hemos obtenido el mismo número con el que empezamos. Asimismo, el primer número, con 1 en su BMS, debería conducir a un número en complemento a 2 con 4 bits y su BMS a 0. No obstante, si examinamos la ecuación con la que obteníamos los posibles valores que podíamos representar en complemento a 2 y con un número de bits dado, podemos observar que no es posible representar el número 8 con la notación de complemento a 2 y 4 bits; por eso no es sorprendente que obtuviéramos un resultado incoherente al intentar obtener el complemento a 2 de este número. Este problema se llama "desbordamiento" *(overflow)*, y se llega a él como resultado de una operación en complemento a 2 cuyo resultado sale fuera del rango de números que pueden representarse con el número de bits utilizado habitualmente. Esto resalta la importancia de que se especifique el número de bits a emplear en las operaciones en complemento a 2.

Cuando se utiliza la notación en complemento a 2, no solamente es importante indicar el número de bits de la representación, sino también que los números estén en complemento a 2 y no en binario normal. Este tipo de problema puede darse cuando se trabaja con varias bases numéricas. ¿Cómo podemos señalar con qué base estamos trabajando? Para ello se emplea una técnica, llamada NOTACION DE SUBINDICES, en la que una letra o un número siguen al número a considerar. Por ejemplo, 123D y 123_{10} quieren decir 123 en decimal.

Las únicas operaciones matemáticas que necesitamos manejar en el sistema binario son la adición y la sustracción, ya que el ordenador fracciona todas sus operaciones aritméticas hasta estos procedimientos simples. Se puede considerar el producto como una adición repetitiva, aunque hay otras maneras de realizar la multiplicación binaria con ordenadores que consideraremos más adelante. De manera análoga puede verse la división como una serie de sustracciones.

Hay un sistema de numeración más que debemos examinar antes de dejar este capítulo, y es un sistema de numeración en base 16. Se llama sistema HEXADECIMAL, y es de vital importancia en el campo de la informática como medio para representar convenientemente números binarios.

Cuando examinamos un número binario tiene muy poco sentido para el ojo humano, pero para el ordenador sí que lo tiene, y mucho. Todos los números que eventualmente introducimos en un ordenador terminan representados en él como dígitos binarios y frecuentemente es útil tener una ligera idea de cómo sería un número en binario sin tener que seguir el proceso tortuoso y largo de escribir una cadena de unos y ceros. El sistema decimal realmente no es apropiado para esta representación, ya que no hay manera, con una representación binaria normal, de convertir fácilmente un número decimal en uno binario. El número decimal 7 es fácil de visualizar, porque proviene de un número binario corto: 0111. Pero, ¿qué hay acerca de 42 ó 146? Realmente no se pueden visualizar con facilidad.

No obstante, en el sistema hexadecimal, de base 16, un solo dígito representa 4 bits. Como su base es 16 necesitaremos algunos caracteres más para representar los dígitos entre 9 y 15. Sencillamente usamos las letras del alfabeto, y por eso los dígitos utilizados en hexadecimal o sistema HEX son 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, y F. A continuación podemos ver la representación binaria de todos estos dígitos:

Hexa-decimal	Binario
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Por eso, es fácil ver que podemos convertir un número de hexadecimal a binario simplemente sustituyendo cada dígito hexadecimal por su equivalente en binario. El número hexadecimal 5E puede convertirse a binario sencillamente escribiendo 0101 en lugar de 5, y 1110 en vez de E, dando el número binario 0101 1110. La adición y la sustracción en el sistema hexadecimal funcionan exactamente del mismo modo que en los sistemas binario y decimal.

En la mayoría de los sistemas de microordenadores, incluyendo el MSX, todos los números se representan en 8 bits, teniendo un valor máximo posible de 1111´1111 en binario o FF en hexadecimal. Los números hexadecimales pueden escribirse como FFH, &FF o $FF, dependiendo de la notación utilizada. Una ventaja más del sistema hexadecimal es que nos permite acortar el número de dígitos necesarios para representar un número de 8 bits, de 3 en decimal a 2 en hexadecimal. Aunque el número mayor que puede manejar el procesador central del ordenador es 255, obviamente pueden representarse otros números más elevados según diferentes tipos de codificación interna. El número de 8 bits ha alcanzado tal importancia en el desarrollo informático que se le ha dado nombre propio: el BYTE. Este, a su vez, puede dividirse en dos números de 4 bits, a los que se les llama metafóricamente NYBBLE.

Finalmente, ¿cómo se convierten números de un sistema de numeración a otro?

Ya hemos considerado la conversión entre hexadecimal y binario; por eso veremos ahora la conversión de números binarios normales a decimal. Si examinamos el número binario 0110, vemos que tiene 0 ochos, 1 cuatro, 1 dos y 0 unidades. Podemos calcular su valor decimal de la manera expuesta a continuación:

$$(0 \times 8) + (1 \times 4) + (1 \times 2) + (0 \times 1) = 6$$

Este proceso puede llevarse a cabo con cualquier número binario normal. La operación inversa, la conversión de decimal a binario, es la más compleja que hemos tratado. Esencialmente es un proceso de divisiones sucesivas. Convirtamos 25 a binario:

25÷2=12	resto	1 LSB del resultado
12÷2=6	resto	0
6÷2=3	resto	0
3÷2=1	resto	1
1÷2=0	resto	1 MSB del resultado

El equivalente binario de 25 es 1 1001, lo que podemos verificar convirtiéndolo de nuevo en decimal (inténtelo usted solo). Observe que los dígitos del número binario aparecen como resto de las divisiones. El proceso continúa hasta que el resultado de la división final es 0, independientemente del valor del resto. El último resto es el BMS. Este método es la norma general para convertir decimales a cualquier otra base: reemplazando el divisor 2 por la raíz de la base del sistema de numeración a la que queremos convertir el número.

* * *

De los sistemas de numeración tratados aquí, su ordenador MSX aceptará enteros, números decimales, hexadecimales, octales[4] y binarios. También utilizará números binarios cuando empiece a escribir programas en código máquina.

Asimismo necesitamos conocer la PRECISION de los números que utilizamos en nuestros programas. La mejor manera de entender la precisión de un número es en términos de su "exactitud". Un número de SIMPLE PRECISION en el sistema MSX tiene seis dígitos, y uno de DOBLE PRECISION, catorce. Para ver lo que significa esto, el número 10 000 001 se representaría en simple precisión como 1E6 (el "E6" representa "diez elevado a seis"). Observe que el dígito menos significativo (el 1) del final del número ha desaparecido. En el formato de doble precisión se puede representar el número completamente como 10 000 001, es decir, con más exactitud. En el sistema MSX, las constantes numéricas se indican posponiéndolas un carácter "!" o conteniendo una E. Por tanto, 10.76! y 1E-3 son ambas constantes de simple precisión. Los números de doble precisión se representan en el sistema con una D, en lugar de una E, intercalada como signo exponencial, o con un "#" al final, o sin ninguna especificación. Así, el número 1.2 estaría representando en la máquina en formato de doble precisión. Otros números en doble precisión son 123.4, 1.234D6 y 1.000345. El formato de doble precisión es el que la máquina toma por defecto y, a menos que se exprese lo contrario, el ordenador tratará a todos los números como de doble precisión[5].

NOTAS DEL CAPITULO

1.—Se trata del sistema SEXAGESIMAL.
2.—En español, también BmS.
3.—En español, también BMS.
4.—Los números octales se basan en un sistema de numeración de base 8. En el sistema MSX se representan precedidos de &O.
5.—Hay un tercer tipo de número: el entero, que se indica con un símbolo "%" al final.

Indice alfabético

El sistema MSX es un concepto radicalmente nuevo que permite a todos los ordenadores personales que utilicen este estándar tener acceso a una amplísima gama de programas y aplicaciones; pero ¿estás preparado para sacar el máximo partido de tu ordenador MSX? ¿Eres consciente de cómo la CPU interacciona con el VDP o el PSG y controla al PPI?

DESCUBRE TU MSX te guiará desde los primeros pasos en la programación en BASIC hasta el dominio absoluto de tu ordenador:

— estructuras de datos,
— programación del *display* de video,
— uso de los *joysticks*,
— sistema de sonido MSX,
— acceso al PPI,
— mapa de memoria MSX,
— programación en lenguaje máquina.

DESCUBRE TU MSX posee un eficaz planteamiento didáctico con multitud de programas ejemplo, explicación detallada de todos los temas, y tablas que resumen los datos interesantes, como posiciones de memoria o variables importantes del sistema.

DESCUBRE TU MSX es un libro indispensable para cualquier poseedor de un microordenador MSX que desee explotar al máximo las posibilidades de su máquina desde el principio; investigando y... ¡descubriendo!

